IBM Watson Research Center
Yorktown Heights, N. Y. 10598
and
The Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

FINAL REPORT - August 31, 1968

JET PROPULSION LABORATORY
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA

A FORMAL THEORY OF CUBICAL COMPLEXES*

by

J. Paul Roth
Eric G. Wagner
Marvin Perlman
Leon S. Levy

IBM Watson Research Center
Yorktown Heights, N. Y. 10598
and
The Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

FINAL REPORT - August 31, 1968

*This paper presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under Contract No. NAS-7-100, sponsored by the National Aeronautics and Space Administration and IBM Corporation, Research Division. Work at IBM Research was performed under Contracts 952341 and 951538.

A FORMAL THEORY OF CUBICAL COMPLEXES

by

J. Paul Roth
Eric G. Wagner
Marvin Perlman
Leon S. Levy

IBM Watson Research Center
Yorktown Heights, N. Y. 10598
and
The Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

ABSTRACT: This report is divided into five parts corresponding roughly to five separate endeavors executed in the study.

Chapter I, entitled "Space Applications of a Minimization Algorithm, describes the logic minimization problem, a slightly updated version of the extraction algorithm, a user's description of the IBM 7094 program MIN6, together with several examples of use of the program in space applications. (This paper will be submitted to the IEEE Transactions on Aerospace & Electronic Systems c/o the Telemetry Editor, John E. Gaffney,Jr., 18100 Frederick Pike, Gaithersburg 20760.)

Chapter II is the paper: "A Calculus and an Algorithm for a Logic Minimization Problem Together with an Algorithmic Notation." This is a writeup of the multiple-output extraction algorithm. In this paper the notion of singular cube and singular complex is introduced, together with a calculus for appropriate computations, together with a new algorithmic notation used to describe the algorithm. A proof of the validity of the algorithm is given. This paper has been submitted to the IBM Journal of Research and Development.

Chapter III is "An Axiomatic Treatment of Roth's Extraction Algorithm." This paper presents a general axiomatic treatment of J. Paul Roth's "extraction algorithm" for the minimization of logical circuits. This treatment brings together the seemingly different versions of the algorithm presented in Roth's different papers, and it provides a general proof of the algorithm over a wide range of cost functions. The minimization problem and the algorithm are presented in an abstract context (i.e., by axioms and without direct reference to any particular application such as switching circuits) and are thus in a form applicable to many "covering problems". Two switching theory applications of the algorithm are sketched at the end of the paper.

Chapter IV, "A Calculus of α-objects, is a description of a very abstract and axiomatic treatment of switching theory, independent of set theory or any other foundational approach. There are two basic operations called catenation and "angle-bracketing". 0 1, x, $\overline{o}$ are primitive objects. Relations, functions, circuits and singular complexes are described in terms of these operations. An algorithm is given for analyzing acyclic logic circuits.

A future area for research is the connection between α-objects and F-notation described in Chapter II.

Chapter V, "An APL Version of MOM the Multiple Ouput 2-Level Minimization Program", describes an APL-implementation of the 2-level MOM program. It follows and conforms to and is based upon the F-notation version described in Chapter II. Several examples of use of this program are included.

SPACE APPLICATIONS OF A MINIMIZATION ALGORITHM

by

J. P. Roth

IBM Watson Research Center
Yorktown Heights, New York

M. Perlman

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

ABSTRACT: This paper is a detailed account of the application of an IBM 7094 minimization program to several design problems at NASA's Jet Propulsion Laboratory of the California Institute of Technology. Specifically these applications are concerned with the design of a curve function generator for a mass spectrometer for a proposed Mars probe and the design of autonomous shift registers with linear and nonlinear feedback, used for classification of binary sequences and counting tasks for spacecraft scientific data processing. The algorithm and program used are first described, followed by a description of the applications.

Space Applications of a Minimization Algorithm

J. P. Roth

IBM Watson Research Center
Yorktown Heights, New York

M. Perlman

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

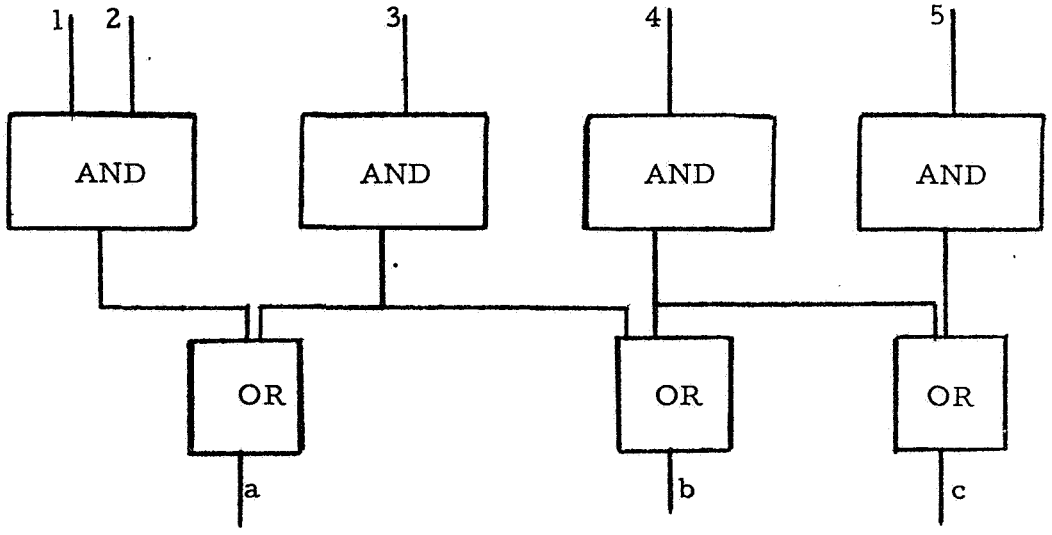
Introduction: The use of minimization algorithms in the synthesis of switching systems is not a new topic but an account in some detail of successful application of these programs to practical problems is of considerable interest. Most of the problems discussed in this paper are concerned with space applications at the Jet Propulsion Laboratory at California Institute of Technology of a program MIN6 of the extraction algorithm [R58] although one of these was connected with the design of the IBM 704 (using an early version MIN 4). Other IBM applications are given elsewhere, e. g. on S/360 [R65].

The problems discussed of the applications are connected with the design of the curve function generator for a mass-spectrometer for a proposed Mars probe and the design of autonomous shift registers with linear and nonlinear logic feedback, used for classification of binary sequences and counting tasks for spacecraft scientific data processing.

The IBM application used MIN 4 to simplify a code translator, converting standard IBM six-bit BCD code to a paper tape code of five bits, with an approximately 50% cost reduction over the original solution.

1. Description of Logic Minimization Problem plus Notation.

A logic circuit of two levels is shown in the figure below. It consists of a level of AND-blocks followed by a level of OR-blocks. It could equally well be a level of NAND blocks followed by a level of NOR-blocks:



1 2 3 4 5	a b c
1 1	1
1 1	1 1
1 1	1 1
1	1

Figure 1. A 2-Level AND/OR Circuit Together With Its Singular Cover.

The programs and algorithms may be interpreted for any pair of "opposite" vertex functions. Below the circuit is a "singular cover" which describes the behavior of the circuit; the first row for example specifies the on-array, specifying that when lines 1 and 2 are 1, the a-output is 1. The absence of symbols in the other columns indicates that the relationship is independent of the values of the "input-variables" 3, 4 and 5 and that, for input-lines 1 and 2 equal to 1, the other outputs, b and c are not determined. A similar interpretation is to be made of the other rows.

This is a brief description of the singular cover notation for describing the behavior of 2-level circuits.

2. Brief Description of Minimization Problem.

Actually, the minimization problem is more general in that so-called "Dont'-Care" conditions are involved; in this event there would be other "cubes" adjoined to the cover: the problem is to find a set of cubes from the "cares" and the "don't-cares" which "cover" the cares and at the same time have a minimum "cost", the cost being some well-defined function of each cover. One such cost is the sum of the number of ones and zeros used in the cover, both for their input and output coordinates. The "cost" relates the cover to the functional realization as in Fig. 1 and its hardware cost.

A related paper [RW68] gives an algorithm for this "multiple-output problem" but a program for this algorithm has not yet been made. A program MIN 6 for an approximation to this minimization problem has received considerable usage within IBM and recently by the Jet Propulsion Laboratory of the

California Institute of Technology. It is the purpose of this paper to describe in some detail these applications, which have some considerable technical importance in themselves.

(MP to furnish description of JPL problems "in capsule".)

The program is based on the extraction algorithm [R59] in its original single-output form, utilizing the Muller-output coding to adapt it to the several-output problem. In general the solutions from such an encoding does not yield a minimum as simple examples show. Indeed the program, as described in the next section, has certain features which allow it to be run in an approximate-minimum form (for purposes of speed and computational feasibility). Consequently for the applications, usually in multiple-output form, a minimum is not ordinarily obtained, but a "sufficiently" good approximation to a minimum is obtained.

3. The Extraction Algorithm.

The extraction algorithm is a means for finding a minimum to the covering problem. It works for the single output case, and is adapted in the program by means of the Muller coding [M54], in the following way.

1. The prime cubes (prime implicants) $\ddot{Z}$ are computed by the # algorithm [ERW 61].
2. The extremals E (members of the core) are computed by the # product [R 58].
3. If E is nonempty, $E \neq \overline{0}$, then E is "extracted" from $\ddot{Z}$ to form Z, $Z = \ddot{Z} - E$, and removed from the care conditions C, $C \leftarrow C \# E$.

3.1 The "less-than" operation is then performed to remove cubes u which can be replaced in any minimum cover by other cubes v. Precisely, u is "less-than" v, $u < v$, if cost (u) $\geq$ cost (v) and v covers of the remaining care conditions, C # E, at least as much as u.

3.2 A new extraction problem is formed consisting of the original C of care conditions diminished by the extremals E which have been computed in 2; if C # E denotes this reduced ensemble of care conditions, then a new set of extremals—call them E_2—is computed according to 2, etc.. The "solution" S is then the "sum" of the E's so computed.

3.3 If at any stage of the computation the "newly computed" ensemble of extremals "vanishes", $E_i = \overline{0}$, then a branch procedure B is invoked which forces a solution S^z by on the one hand selecting a cube z (by some elaborate process) and treating it as if it were an extremal and on the other "rejecting" z (as if it were < some other cube), to obtain a solution $S^{\overline{z}}$. That which has lower cost constitutes a minimum for the original problem.

This is a slightly updated version of the programmed algorithm MIN 6, whose use and some applications thereof is defined below.

4. THE MIN-6 PROGRAM

4.1 Background

The MIN-6 program was written for the IBM 7094 general purpose computer to determine a K-cover of L of minimum or approximate minimum cost. K denotes a cubical complex containing the subcomplex L. The vertices N = K - L are the unspecified or don't care vertices. The program is based on J. Paul Roth's extraction algorithm [R59] for single-output Boolean functions. The multi-output problem is first converted to an imaginary single-output problem by Muller coding. The extraction algorithm is then applied. See Section 3. The minimization of the single output function yields the simultaneous minimization of the Boolean functions representing the original multi-output problem [M54]. A K-cover of L is any collection of cubes C such that each vertex of L is contained in some cube of C. Cost is defined as the number of diodes required in a two-level AND-OR implementation. When considering combinational logic networks utilizing large scale integrated circuits (LSI), cost can be defined as the number of interconnections.

The MIN-6 program consists of three steps. During step 1 the input data is read and an array of prime cubes is derived. A cube z of K is a prime cube if $\delta_i z = \emptyset$ for all i. A prime cube corresponds to a prime implicant in Quine's terminology when K = L. Every K-cover of L of minimum cost is contained in the set of prime cubes. In step 2 prime cubes are selected to form a K-cover of L of minimum cost. During step 3 a solution (or several solutions of equal or near equal cost) are written out.

The MIN-6 options fall into two categories. These are (theoretically) minimum-cost solutions and approximate minimum-cost solutions. The difference is in the performance of step 1. For minimum-cost solutions the entire array of prime cubes is derived by means of the "sharp" algorithm [R58]. For approximate minimum-cost solutions only a portion of the prime cubes is derived for a given problem by means of the "coface" algorithm [R59]. In larger problems the sharp algorithm can result in overflow in core or the failure to find a solution in a reasonable running time. Since the coface algorithm derives many fewer prime cubes, it requires less running time and has considerably less chance for overflow than the sharp algorithm. Furthermore and more importantly, step 2 where the selection of a subset of prime cubes is made runs much faster after cofacing than after sharping.

There is no known method for predicting the running time for either category of options for a given problem. If experience with a given type of problem indicates that no more than three solutions are extracted in a reasonable time by sharping, then cofacing should be used.

4.2 The Sharp Algorithm

Given the cubes $u = (u_1, u_2, \ldots, u_n)$ and $v = (v_1, v_2, \ldots, v_n)$, the sharp product $u \# v$ are the vertices of u that are not in v. The resulting set of vertices are represented as a cube or the union of cubes of largest possible dimension. The coordinate representation of a cube is an n-tuple where each component is a 0, 1, or X. The dimension of a cube is equal to the number of X's (free coordinates). The cost of a cube is equal to n minus the number of X's (i.e., the number

of bound coordinates). The #-product of coordinates appears in TABLE 4-1.

u_i \ v_i	0	1	X
0	z	y	z
1	y	z	z
X	1	0	z

TABLE 4-1 THE #-PRODUCT OF COORDINATES

The sharp product u # v is determined from the #-product of coordinates as follows:

$$u \# v = \begin{cases} u \text{ if } u_i \# v_i = y \text{ for any } i \\ \emptyset \text{ if } u_i \# v_i = z \text{ for all } i \\ \sum_i (u_1, \ldots, u_{i-1}, \alpha_i, u_{i+1}, \ldots, u_n) \\ \quad \text{where } u_i \# v_i = \alpha_i = 0 \text{ or } 1 \end{cases}$$

In the third case, the logical summation runs over all i where $u_i \# v_i = 0$ or 1.

EXAMPLE 1

a. XXX # 01X = 1XX + X0X

b. X10 # XX1 = X10

c. 10X # 1XX = $\emptyset$

The sharp product is non-commutative and non-associative. It does however satisfy the following distributive law

$$(u + v) \# w = (u \# w) + (v \# w)$$

Other properties follow from the definition:

1. $u \# v = \emptyset$ if $u \subseteq v$
2. $u \# v \subseteq u$
3. $(u \# v) \# w = (u \# w) \# v$

The sharp algorithm is used in MIN-6 to derive all the prime cubes of the on or off array. Minimize on (disjunctive minimum) and minimize off (conjunctive minimum) are MIN-6 options which fall into the category of minimum-cost solutions. If the on array (input cubes which result in a 1 output) are supplied to the computer and a minimize on is requested, the sharp algorithm is used twice. First the on array is sharped from the universal cube. This yields the prime cubes of the off array. The prime cubes of the off array are then sharped from the universal cube to obtain the prime cubes of the on array. The double sharp routine can be avoided by giving the computer the off array (on array) when requesting a minimize on (minimize off) option.

EXAMPLE 2

Given

A B C	f (A, B, C)
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	1

To realize a minimize on solution the off array {000, 010, 100, 101} is sharped from the universal cube XXX as follows:

(((XXX # 000) # 010) # 100) # 101

1. XXX # 000 = 1XX + X1X + XX1

2. $(1XX + X1X + XX1) \# 010 = 1XX + 11X + X11 + XX1$
$= 1XX + XX1$

3. $(1XX + XX1) \# 100 = 11X + 1X1 + XX1$
$= 11X + XX1$

4. $(11X + XX1) \# 101 = 11X + 0X1 + X11$

The prime cubes of the on array are {11X, 0X1, X11}. Note that in step 2 11X c 1XX and X11 c XX1. Similarly, in step 3, 1X1 c XX1. Property 1 of the sharp product can be used to determine whether a cube is contained in a higher dimensional one. This corresponds to a <-operation where the cube of lower dimension (a less than) is discarded since it is contained in one of higher dimension (hence, of lower cost).

4.3 The Coface Algorithm

Both the on and off array must be supplied to the computer before cofacing. The dimension of each cube in the on array is increased as much as possible without overlapping the off array. The procedure is to replace each cube of the on array with a prime cube which contains it, but does not contain any vertices of the off array. Essentially the first bound coordinate of a given cube in the on array is replaced by a free coordinate. The higher dimensional cube is then tested to see if it belongs to the K complex. If it does, the given cube is replaced by the higher dimensional one. This is repeated for each of the remaining bound coordinates. The resulting cube will always be a prime cube. In general, only a portion of all possible prime cubes are derived by this method and any cover selected from these prime cubes will not be a minimum-cost cover. Historically, this procedure was explained in terms of a succession of face and coface

operations [R59], hence the term cofacing was used. The cofacing algorithm can be implemented with sharping. The off array is sharped from a given cube in the on array after its dimension has been incremented. The given cube will be unaltered if it does not contain any vertex of the off array.

EXAMPLE 3

Given

A B C	f (A, B, C)
0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	-
1 1 0	1
1 1 1	0

The dash (-) denotes an unspecified or don't care condition. The on and off arrays are listed as follows:

ON ARRAY	OFF ARRAY
0 0 0	0 1 0
0 0 1	0 1 1
1 0 0	1 1 1
1 1 0	

Tests for 0 0 0

((X00 # 101) # 011) # 111 = X00

XX0 # 010 ≠ XX0

((X0X # 010) # 011) # 111 = X0X

Therefore 000 is replaced with X0X, a prime cube contained in K.

4.4 Computation of Extremals and Less Thans

A prime cube z_i of K is termed an L-extremal [R59] if it contains a vertex of L not contained in any other prime cube of K. The set E of all L-extremals of K must be contained in any minimum-cost cover of L. When K = L, E corresponds to the "core" in Quine's terminology. The removal of extremals reduces the number of prime cubes which must be selected to form a cover. Let Z equal the prime cubes from which a K cover of L is to be selected. Sharping is performed to determine whether z_i is an extremal. The prime cube z_i is an extremal if and only if

$$z_i \# \{Z - z_i\} \neq \emptyset$$

After identifying and storing the extremals, the remaining prime cubes are partially ordered according to dimension. Each <-maximal (i.e., less than) prime cube contained in one of higher (or equal) dimension is discarded. The remaining prime cubes are subjected to the same process since the removal of less-thans may introduce another set of extremals. The process for some problems continues until the remaining set of prime cubes is empty. In this case the union of all the ordered sets of extremals $\{E_1, E_2, \ldots, E_r\}$ is a unique minimum-cost K-cover of L.

For many problems, however, a point is reached where all of the remaining prime cubes are maximal under the <-operation (i.e., none are less thans) and none are extremals (i.e., each remaining vertex is covered by more than one prime cube). When both of these conditions hold (K_r, L_r) is termed irreducible. K_r is the complex resulting from alternately removing i^{th} ordered extremals E_i and applying the <-operation for all $i \leq r - 1$. L_r is a subcomplex of K_r. If K = L and (K, L) is irreducible, the

cubical complex represents a cyclic Boolean function. Whenever a point is reached where (K_r, L_r) is irreducible, the MIN-6 program goes into a branching mode.

4.5 Branching Mode

Branching starts with a selection of a cube u in K_r. First n is treated as if it were an extremal. The <-operation is applied to $\{K_r - u\}$. This subcover is then tested for new extremals and the extraction algorithm continues. If no additional branch points arise, a K-cover of L will be found which contains u. Its cost is computed and stored. The program returns to the branch point where (K_r, L_r) is irreducible. Then u is treated as if it were a less than some other cube in K_r. This subcover is then tested for new extremals and the extraction algorithm continues. No additional branch points arise and a K-cover of L will be found which does not contain u. Its cost is computed and compared with that of the solution containing u. The lower cost solution (or either one if the costs are equal) is a minimum cost solution.

For many problems more than one branch point appears in searching for a minimum-cost solution. At each branch point, a "best" cube is selected and put into a solution buildup as an extremal. This continues until a solution is reached. Successive branch points or nodes may be diagrammed as end points of a branch of a tree. After a solution and its cost is computed and stored, the program returns to the last branch point. The "best" cube selected at this point is then treated as a less than in forming another solution in which this "best" cube is not included. If the cost of this solution is cheaper than (or

the same as) the previous one, it is retained as a currently cheapest solution (CCS). The program continues to retrace branches to a previous level of branching, return on an alternate branch, and proceed toward a new solution. Branch tracing is interrupted when it has been determined that the path can only lead to a more costly solution than the CCS. A branching tree is illustrated in Fig. IV-1.

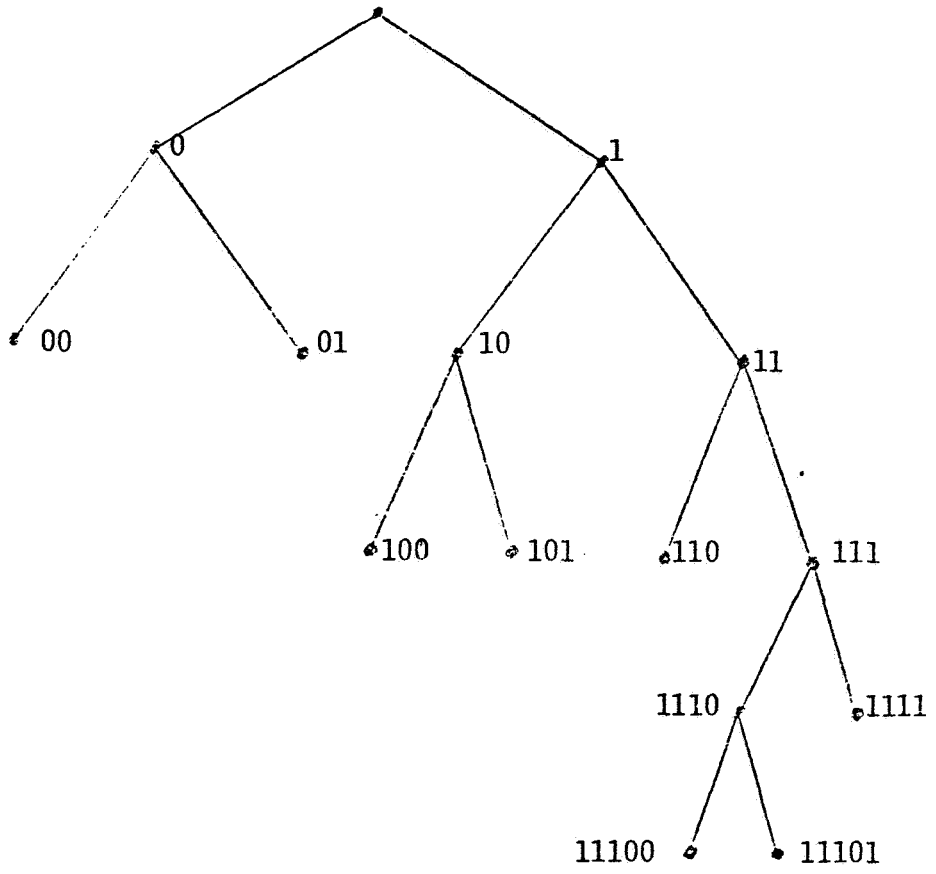


Figure 4-1. A Min-6 Branching Tree.

The cost, branch number, branch level, and a binary sequence which identifies the location in the tree representation of branching of every CCS is written out. The branch number of a solution refers to the number of terminal branches as read from right to left up to and

including the CCS it identifies. Each terminal is associated with a solution. The branch level is equal to the number of bits in the binary sequence. At each branch point a bit is added to the right of a binary sequence. A 1 (0) is added if the "best" cube at the previous branch point was treated as an extremal (less than). From Fig. 4-1, the solution associated with 100 has a branch number 6 and a branch level 3.

The selection of a cube when branching should be one that:

1. minimizes the total extraction time and
2. lowers the cost at which subsequent branches can be terminated.

A cube that yields a large number of new prime cubes for exclusion or inclusion in a solution satisfies part 1. If this leads to a low-cost solution at the end of the branch being traced, part 2 is satisfied. Selection criteria is a current research problem.

MIN-6 is implemented to select a cube whose "crown" has the greatest dimension. The crown of a given cube is defined as the subcube of the smallest dimension that contains all the "care" vertices of the given cube.

5. DESCRIPTIONS OF MINIMIZED DESIGNS

5.1 Feedback Shift Register Code Translator

A generalized feedback shift register (FSR) appears in Fig. 5-1. The content of the $i^{\underline{th}}$ stage (a two-state memory element) at clock pulse interval (CPI) k is denoted as a_{k-i}. The bit being fed back during CPI k is a Boolean function of the states of the r stages. Hence

$$a_k = f(a_{k-1}, a_{k-2}, \ldots, a_{k-r}) \tag{5.1}$$

The state of the $i^{\underline{th}}$ stage at CPI k becomes the state of the $(i + 1)^{\underline{th}}$ stage at CPI k + 1.

$$a_{k-i} = a_{(k + 1) - (i + 1)}$$

The initial state of the $i^{\underline{th}}$ stage is represented as a_{-i} where k = 0.

The FSR is in a subclass of autonomous finite state machines. The sequence $\{a_k\}$ is periodic and the length of the period ℓ is always dependent upon the feedback function and may depend upon the initial state of the register.

EXAMPLE 5-1

$$a_k = a_{k-1}\, a_{k-3} \oplus a'_{k-4}$$

where ($\oplus$) denotes sum modulo 2 (i.e., EXCLUSIVE-OR) and (') denotes complementation. Logical multiplication is denoted by juxtaposition. Successive states $a_{k-1}\, a_{k-2}\, a_{k-3}\, a_{k-4}$ and a_k are tabulated as follows:

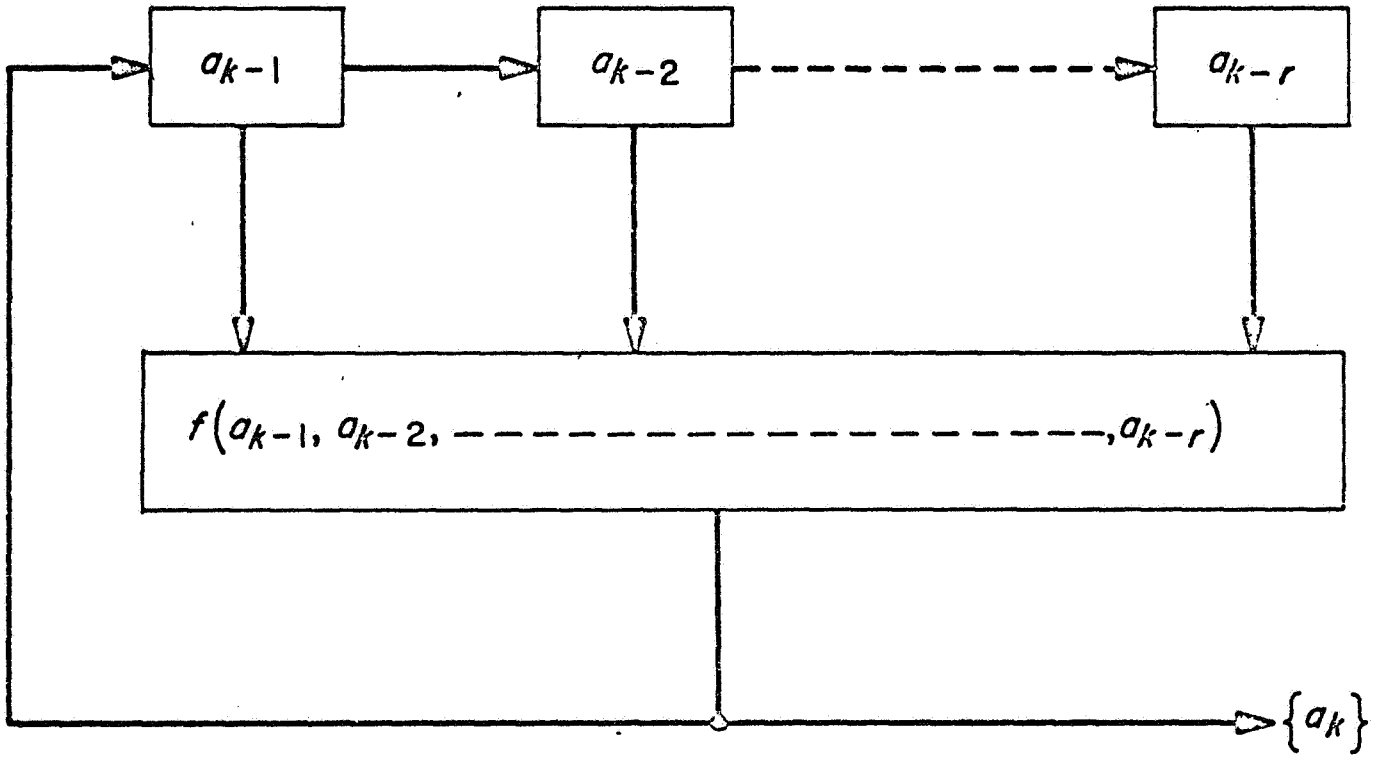


FIG. 5-1

GENERALIZED FEEDBACK SHIFT REGISTER

k	a_{k-1}	a_{k-2}	a_{k-3}	a_{k-4}	a_k
0	0	0	0	0	1
1	1	0	0	0	1
2	1	1	0	0	1
3	1	1	1	0	0
4	0	1	1	1	0
5	0	0	1	1	0
6	0	0	0	1	0
0	0	0	1	0	1
1	1	0	0	1	0
2	0	1	0	0	1
3	1	0	1	0	0
4	0	1	0	1	0
0	0	1	1	0	1
1	1	0	1	1	1
2	1	1	0	1	0
0	1	1	1	1	1

The feedback function decomposes the 2^4 states into branchless cycles of length 1, 3, 5, and 7 as shown in the state diagram of Fig. 5-2. States are labeled with their decimal equivalents.

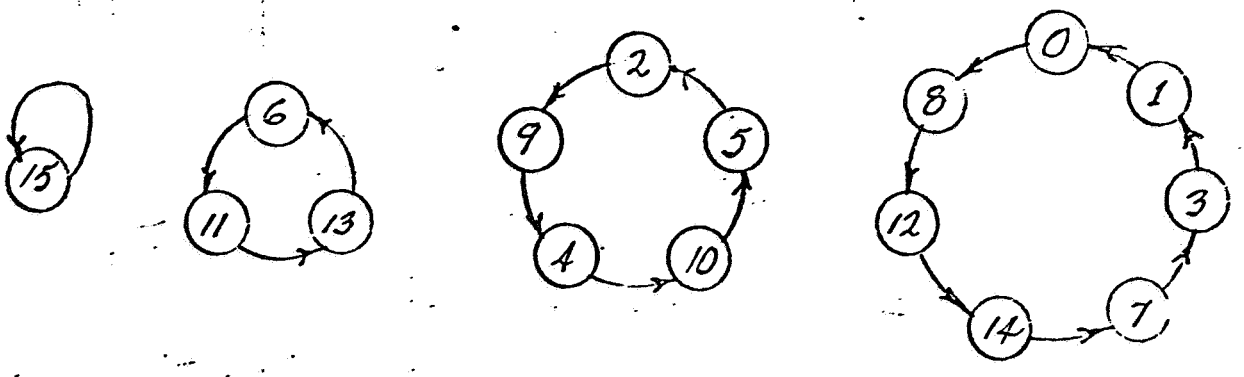


FIG. 5-2 FSR STATE DIAGRAM FOR $a_k = a_{k-1}\, a_{k-3} \oplus a'_{k-4}$

A necessary and sufficient condition for distinct states to have distinct successors [G-67] is that a_k be of the form shown in 5.2.

$$a_k = g(a_{k-1}, a_{k-2}, \cdots, a_{k-r+1}) \oplus a_{k-r} \qquad (5.2)$$

In example 5.2

$$a_k = (a'_{k-1} + a'_{k-3}) \oplus a_{k-4}$$

and branchless cycles result.

The FSR has numerous application in addition to serving as a mathematical model for random number generation, finite state machines, and Markovian processes [G-67]. Applications include counting, scaling, error-correcting code generation and detection, ranging, prescribed sequence generation, and single-valued curve generation.

Among the 2^{2^r} switching functions of r Boolean variables, there are $2 \cdot \phi(2^r-1)/r$ linear functions which result in cycles of length 2^r-1. [$\phi(n)$, the Euler-phi-function, is the number of integers no greater than n that are relatively prime to n]. These are termed maximal-length cycles. A switching function which can be expressed as

$$f(x_1, x_2, \cdots, x_n) = c_o \oplus c_1x_1 \oplus \cdots \oplus c_nx_n \qquad (5.3)$$

is linear where $c_i = 0$ or 1 for $0 \leq i \leq n$. When the feedback function is linear, a necessary but not sufficient condition for realizing maximal-length cycles is that the content of an even number of stages is fed back. For many values of r, as few as two stages are required. Two-tap linear logic feedback for an r-stage FSR yields the most efficient FSR (cycle length per cost of combinational logic) in terms of implementation. The maximal-length sequence associated with a linear FSR also has useful pseudo-randomness characteristics including a two-level autocorrelation property [G-67]. The simplicity of the two-tap linear

FSR, its serial character, and synchronous behavior makes it attractive for science data processing tasks in interplanetary spacecraft.

Successive states do not correspond to linearly increasing (or decreasing) binary numbers. Serial techniques involving another FSR can be used to decode or translate a count. In many cases, however, a parallel translation is required in the interest of speed. No analytical solution has been found for transforming successive states of an FSR with long cycle lengths to ordered binary numbers which are in a one-to-one correspondence.

MIN-6 enables a logical designer to minimize a two-level AND-OR diode matrix which serves to translate an FSR code to a binary number. This is illustrated in example 5.2.

EXAMPLE 5.2

Given a 4 stage FSR with the following feedback function:

$$a_k = a_{k-1} \oplus a_{k-4}$$

Every non zero state lies in a maximal-length cycle of length 15. Let the initial state $a_{-1}\ a_{-2}\ a_{-3}\ a_{-4}$ of 1 1 0 0 represent a binary 0. Successive states are to represent the binary numbers from 1 through 14 respectively. The FSR state 0 0 0 0 is singular (i.e., lies in a cycle of length 1) and is treated as a "don't care." The MIN-6 solution of Example 5.2 appears in Fig. 5.2. Fig. 5.2 is a photo reduction of the actual off-line printout of 5 pages. The canonical input array (upper left) appears on page 1. The canonical input array option defaults to a minimize on solution unless minimize off is specified. The input cubes are supplied to the computer in octal whereas the output cubes are supplied in binary. Note that the digit 2 represents a don't

TEST 2 FSR TO BINARY TRANSLATOR ENS 1100

```
PROBLEM HAS   4 INPUTS AND  4 OUTPUTS

CANONICAL INPUT ARRAY
          INPUTS                                   OUTPUTS
      14    1100                                      0000
      16    1110                                      0001
      17    1111                                      0010
      07    0111                                      0011
      13    1011                                      0100
      05    0101                                      0101
      12    1010                                      0110
      15    1101                                      0111
      06    0110                                      1000
      03    0011                                      1001
      11    1001                                      1010
      04    0100                                      1011
      02    0010                                      1100
      01    0001                                      1101
      10    1000                                      1110
      00    0000                                      2222
    END OF ARRAY
```

```
PROBLEM HAS   4 INPUTS AND  4 OUTPUTS

OFF ARRAY

ARRAY HAS   32 CUBES OF COST   160

            1100                                      1000
            1110                                      1000
            1111                                      1000
            0111                                      1000
            1011                                      1000
            0101                                      1000
            1010                                      1000
            1101                                      1000
            1100                                      0100
            1110                                      0100
            1111                                      0100
            0111                                      0100
            0110                                      0100
            0011                                      0100
            1001                                      0100
            0100                                      0100
            1100                                      0010
            1110                                      0010
            1011                                      0010
            0101                                      0010
            0110                                      0010
            0011                                      0010
            0010                                      0010
            0001                                      0010
            1100                                      0001
            1111                                      0001
            1011                                      0001
            1010                                      0001
            0110                                      0001
            1001                                      0001
            0010                                      0001
            1000                                      0001
    END OF ARRAY
```

```
PROBLEM HAS   4 INPUTS AND  4 OUTPUTS

DONT CARE ARRAY

ARRAY HAS    1 CUBES OF COST     8

            0000                                      1111
    END OF ARRAY
```

```
ON ARRAY

   49 CUBES

COMPLETE ARRAY HAS   23 CUBES OF COST   108

ELAPSED TIME AT START OF EXTRACTION IS    0.511

BRANCHING HAS OCCURRED. ELAPSED TIME IS    0.531

PRE-BRANCHING EXTREMALS -    6      CUBES REMAINING -   13

LATEST SOLUTION HAS   12 CUBES OF COST    56
ELAPSED TIME IS    0.551
BRANCH NUMBER -   1    BRANCH LEVEL -  4
1111

LATEST SOLUTION HAS   13 CUBES OF COST    55
ELAPSED TIME IS    0.581
BRANCH NUMBER -   3    BRANCH LEVEL -  3
110

LATEST SOLUTION HAS   13 CUBES OF COST    53
ELAPSED TIME IS    0.616
BRANCH NUMBER -   4    BRANCH LEVEL -  2
10

LATEST SOLUTION HAS   12 CUBES OF COST    53
ELAPSED TIME IS    0.650
BRANCH NUMBER -   5    BRANCH LEVEL -  2
01

MAXIMUM NUMBER OF BRANCHING LEVELS USED WAS  4

NUMBER OF BRANCHES TRACED -   5

EXTRACTION TIME -    0.168
```

```
COST OF SOLUTION BEFORE REMOVING REDUNDANT OUTPUT LINES -   53

SOLUTION NUMBER  1 HAS   13 CUBES OF COST   50

INPUTS                                        OUTPUTS
1110                                            01   0001
2101                                            05   0101
1201                                            02   0010
1012                                            04   0100
0201                                            04   0100
2111                                            02   0010
1020                                            02   0010
0200                                            03   0011
2002                                            10   1000
0221                                            01   0001
0022                                            10   1000
0220                                            10   1000
2020                                            04   0100
END OF ARRAY

COST OF SOLUTION BEFORE REMOVING REDUNDANT OUTPUT LINES -   53

SOLUTION NUMBER  2 HAS   12 CUBES OF COST   50

INPUTS                                        OUTPUTS
1101                                            07   0111
1110                                            01   0001
2111                                            02   0010
0201                                            04   0100
1020                                            02   0010
1002                                            12   1010
0200                                            03   0011
1012                                            04   0100
0221                                            01   0001
0022                                            10   1000
0220                                            10   1000
2020                                            04   0100
END OF ARRAY

ELAPSED TIME =   0.929 MINUTES
```

FIG. 5-2

FSR (4-stage) Code
to Binary Translator

care coordinate. A preprocessor converts the input octal array to a binary array. On page 2 (upper right) the don't care array is separated out by the preprocessor. On page 3 (center left) the Muller coded off array is determined by the preprocessor. Note that a minimized on is actually being requested. Therefore, the off array is sharped from the universal cube to generate the prime cubes of the on array. The 32 cubes with a cost of 160 diodes in the off array is associated with Muller coded off array.

The 4-input 4-output problem is hereafter treated as an imaginary 8-input 1-output network. The input cube and its associated Muller coded output cube are combined to form a vertex of eight coordinates. The Muller transformation also introduces "don't care" vertices [M-66]. On page 4 (center right), the number of cubes (23) and their cost (108 diodes) of the complete array is given. These represent a K-cover of L of the 8-input 1-output problem resulting from sharping the Muller coded off array from the universal cube. Elapsed times appearing in the printout are in minutes. The extraction algorithm, for example, was applied 0.511 minutes after the problem was received. The branching status of each currently cheapest solution is printed out. The maximum number of branching levels and the number of branches traced are noted. Page 5 (bottom of Fig. 5-2) gives two of the four solutions found. These are of equal cost. However, solution 2 requires one less cube or gate. The total elapsed time was 0.929 minutes of which 0.168 minutes was consumed by the extraction algorithm.

Let x_i correspond to a_{k-i}. The output $z_1\ z_2\ z_3\ z_4$ is the translation into binary of the FSR code represented by $x_1\ x_2\ x_3\ x_4$.

Solution 2 of example 5.2 is expressed algebraically as follows:

$$z_1 = \underline{x_1\, x_2'\, x_3'} + x_1'\, x_2' + x_1'\, x_4'$$

$$z_2 = \underline{x_1\, x_2\, x_3'\, x_4} + x_1'\, x_3'\, x_4 + x_1\, x_2'\, x_3 + x_2'\, x_4'$$

$$z_3 = \underline{x_1\, x_2\, x_3'\, x_4} + x_2\, x_3\, x_4 + x_1\, x_2'\, x_4' + \underline{x_1\, x_2'\, x_3'} + \underline{x_1'\, x_3'\, x_4'}$$

$$z_4 = \underline{x_1\, x_2\, x_3'\, x_4} + x_1\, x_2\, x_3\, x_4' + \underline{x_1'\, x_3'\, x_4'} + x_1'\, x_4$$

The underlined terms are shared. The derivation of the simultaneously minimized functions from the minimized single function (after the Muller transformation) is detailed in [M-54]. The second level of minimization is approximate for many multiple output problems since the search for a minimum-cost solution would require the generation of an unusually large number of prime cubes after Muller coding [M-66]. The cost of the inputs to the first level of gating is minimized and any redundancy in their outputs is removed when forming the second level of gating. The MIN-6 solutions gives the cost before and after removing redundant output lines from their outputs. See Fig. 5-2. Solutions 1 and 2 of Example 5.2 have a cost of 50 diodes. The canonical form requires 88 diodes. This represents a reduction of 43%.

EXAMPLE 5-3

An FSR to binary translator for a 5-stage maximal length FSR was minimized with MIN-6. The feedback function was

$$a_k = a_{k-2} \oplus a_{k-5}$$

Every non-zero state lies in a maximal-length cycle of length 31. A total of 31 state assignments were minimized with the coface algorithm. Each of the 31 cyclic permutations of the FSR states were put into a

```
TEST 11 FSR TRANSLATOR  INS 01101

ON ARRAY

  129 CUBES

COMPLETE ARRAY HAS   66 CUBES OF COST   376

ELAPSED TIME AT START OF EXTRACTION IS    0.585

BRANCHING HAS OCCURRED. ELAPSED TIME IS    0.669

PRE-BRANCHING EXTREMALS -   12      CUBES REMAINING -   32

LATEST SOLUTION HAS   26 CUBES OF COST   148
ELAPSED TIME IS    0.737
BRANCH NUMBER -   1    BRANCH LEVEL -  7
1111111

LATEST SOLUTION HAS   25 CUBES OF COST   143
ELAPSED TIME IS    0.772
BRANCH NUMBER -   2    BRANCH LEVEL -  7
1111110

LATEST SOLUTION HAS   25 CUBES OF COST   143
ELAPSED TIME IS    0.818
BRANCH NUMBER -   9    BRANCH LEVEL -  7
1110101

LATEST SOLUTION HAS   26 CUBES OF COST   140
ELAPSED TIME IS    0.930
BRANCH NUMBER -  26    BRANCH LEVEL -  4
1010

MAXIMUM NUMBER OF BRANCHING LEVELS USED WAS 10

NUMBER OF BRANCHES TRACED -   152

EXTRACTION TIME -    1.255
```

```
COST OF SOLUTION BEFORE REMOVING REDUNDANT OUTPUT LINES -  140

SOLUTION NUMBER  1 HAS   26 CUBES OF COST  132

INPUTS                                    OUTPUTS
11201                                        12   01010
02000                                        06   00110
12101                                        20   10000
10211                                        10   01000
00102                                        02   00010
01210                                        20   10000
00201                                        04   00100
11012                                        32   11010
10121                                        20   10000
21001                                        01   00001
11112                                        06   00110
20100                                        22   10010
00012                                        02   00010
00210                                        01   00001
01120                                        14   01100
21011                                        06   00110
10112                                        15   01101
12021                                        02   00010
12200                                        04   00100
20121                                        11   01001
02002                                        20   10000
01212                                        10   01000
20020                                        20   10000
12121                                        01   00001
12102                                        01   00001
10202                                        01   00001
END OF ARRAY

ELAPSED TIME =   2.120 MINUTES
```

FIG. 5-3

FSR (5-stage) Code to
Binary Translator

one-to-one correspondence with the 5 place binary numbers from 0 0 0 0 0 through 1 1 1 1 0. The canonical form for each assignment consists of 230 diodes. Those assignments which yielded a reduction of more than 35% were rerun with the sharp algorithm.

The assignment having 0 1 1 0 1 as the initial state yielded the highest reduction, namely 42.6%. See Fig. 5-3.

5.2 Prescribed Sequence Generator

Serial data emanating from a digital data processor in a spacecraft are divided into blocks or frames. Binary sequences are inserted to identify the beginning of a data frame.

Every n-bit sequence which is subperiod free can be characterized as a binary (n, r) ring sequence. The (n, r) BRS is an ordered cycle of n distinct r-bit subsequences. Any value of r which yields n distinct subsequences may be used. Necessarily $2^r \geq n$. The sequence

a_5	a_4	a_3	a_2	a_1	a_0
0	0	0	1	0	1

is represented as a (6, 4), (6, 5), (6, 6) and (6, 7) binary ring sequence (BRS) in Table 5-1.

(6, 4)	(6, 5)	(6, 6)	(6, 7)
0 0 0 1	0 0 0 1 0	0 0 0 1 0 1	0 0 0 1 0 1 0
1 0 0 0	1 0 0 0 1	1 0 0 0 1 0	1 0 0 0 1 0 1
0 1 0 0	0 1 0 0 0	0 1 0 0 0 1	0 1 0 0 0 1 0
1 0 1 0	1 0 1 0 0	1 0 1 0 0 0	1 0 1 0 0 0 1
0 1 0 1	0 1 0 1 0	0 1 0 1 0 0	0 1 0 1 0 0 0
0 0 1 0	0 0 1 0 1	0 0 1 0 1 0	0 0 1 0 1 0 0

Table 5-1 BRS Representations of 000101

A (6, 3) BRS characterization does not exist for 000101 even though $2^3 > 6$. This is due to the double appearance of the subsequence 010 in the ring. Successive states of an r-stage FSR can be made to correspond to n successive r-bit subsequences in the (n, r) BRS. The minimum value of r which characterizes an n-place subperiod free sequence as an

(n, r) BRS falls in the range of values expressed by 5.3.

$$1 + \left[\log_2 n\right] \leq r \leq n-1 \tag{5.3}$$

The bracketed term denotes the nearest integer which is less than $\log_2 n$.

A proof is given in [Y62] of the existence of (n, r) BRSs for any r and $n \leq 2^r$. If only the length n is specified one may be found with a BRS characterization where r has the smallest possible value which satisfies the inequalities of 5.4.

$$2^{r-1} < n \leq 2^r \tag{5.4}$$

All n-bit subperiod free sequences for $1 < n \leq 9$ are classified according to the feedback function of their (n, r_{min}) BRS generators in [P68].

A constructive proof appears in [G67] showing that the linear feedback function of an r-stage maximal length FSR can be altered to realize any cycle length from 1 to 2^r. The structure of the resulting sequence, however, is fixed. In general, an altered maximal-length sequence must be transformed to the desired sequence. This can be done by an r x 1 AND-OR matrix which translates ℓ successive r-bit states to the desired sequence of length ℓ. Note that sequences with subperiods can also be derived in this manner.

When designing a prescribed sequence generator, (n, r) BRS generation can be compared on the basis of overall cost for various values of r. Overall cost includes cost of memory elements as well as decision elements (i.e., combinational logic). These results can then be contrasted with a maximal-length FSR generator altered, if necessary,

to cycle through n successive states each of which is transformed to a single bit in the desired sequence. These steps are illustrated in the following example.

EXAMPLE 5-4.

The sequence

$$0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1 \quad \{a_k\}$$

$$9\ 8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0 \quad k$$

is to be generated. The $(n, r_{min.})$ corresponding to $\{a_k\}$ is (10, 4). The minimized feedback functions for a (10, 4) and a (10, 5) BRS generator are determined from Table 5-2.

k	a_{k-1}	a_{k-2}	a_{k-3}	a_{k-4}	a_{k-1}	a_{k-2}	a_{k-3}	a_{k-4}	a_{k-5}	a_k
0	0	1	0	0	0	1	0	0	0	1
1	1	0	1	0	1	0	1	0	0	1
2	1	1	0	1	1	1	0	1	0	1
3	1	1	1	0	1	1	1	0	1	0
4	0	1	1	1	0	1	1	1	0	1
5	1	0	1	1	1	0	1	1	1	0
6	0	1	0	1	0	1	0	1	1	0
7	0	0	1	0	0	0	1	0	1	0
8	0	0	0	1	0	0	0	1	0	1
9	1	0	0	0	1	0	0	0	1	0

Table 5-2 State Tables for a (10, 4) and a (10, 5) BRS Generator for 0100010111

Unused entries are treated as don't cares.

(10, 4) BRS generator

$$a_k = a'_{k-1}\ a'_{k-2}\ a'_{k-3} + a'_{k-1}\ a_{k-2}\ a_{k-3} + a_{k-2}\ a'_{k-3}\ a'_{k-4} + a_{k-1}\ a_{k-2}\ a_{k-4} + a_{k-1}\ a'_{k-2}\ a_{k-3}\ a_{k-4}$$

cost $5\delta + 21\gamma$

(10, 5) BRS generator

$$a_k = a'_{k-5}$$

cost $5\delta + 0\gamma$

(The reader may verify that branches occur in the states of the (10, 4) BRS and the 1111 state is singular.) The cost of a memory element is denoted as δ. The decision element is assumed to be a diode gate with a cost γ equal to the number of diodes. The costs of the (10, 4) and the (10, 5) BRS generators are equal when

$$5\delta = 4\delta + 21\gamma \text{ or } \delta/\gamma = 21$$

The feedback function

$$b_k = b_{k-1} \oplus b_{k-4} \oplus b'_{k-1}\ b_{k-2}\ b_{k-3}\ b_{k-4}$$

has a major cycle of length 10. The maximal-length cycle 15 associated with $b_{k-3} \oplus b_{k-4}$ is shortened by skipping 5 states. The nonlinear term $b'_{k-1}\ b_{k-2}\ b_{k-3}\ b_{k-4}$ causes the state 0111 to be succeeded by 0011 instead of 1011 by inverting the bit that is normally fed back. By treating the 5 states that are skipped and the singular state 0000 as don't cares, the feedback function reduces to

$$b_k = b_{k-1}\ b'_{k-4} + b'_{k-1}\ b'_{k-2}\ b_{k-4}$$

The state diagram appears in Fig. 5-4.

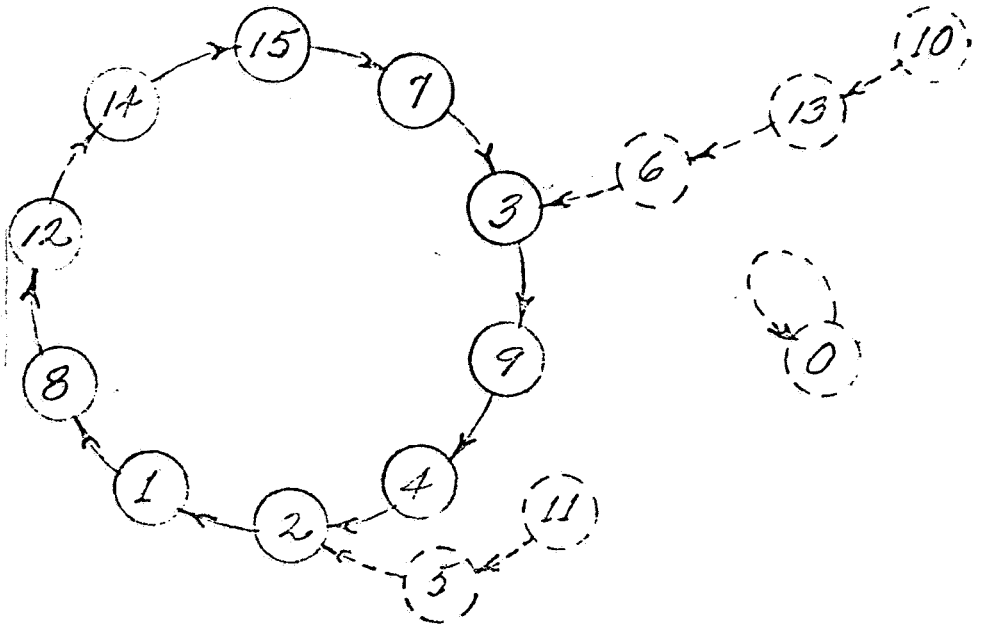


Fig. 5-4 . FSR State Diagram for

$$b_k = b_{k-1}\, b'_{k-4} + b'_{k-1}\, b'_{k-2}\, b_{k-4}$$

Though $\{b_k\}$ has the required cycle length when properly initialized, none of the 10 possible initial states yields the desired sequence $\{a_k\}$. The distinct successive states $b_{k-1}\, b_{k-2}\, b_{k-3}\, b_{k-4}$ can be transformed to bits corresponding to $\{a_k\}$. Each of the 10 cyclic permutations of the 10 states can be used in the state assignment for realizing $\{a_k\}$. Two of the 10 require combinational logic of minimum cost. A minimum cost assignment appears in Table 5-3 where $b_{-1}\, b_{-2}\, b_{-3}\, b_{-4}$ of 1110 corresponds to a_o.

k	b_{k-1}	b_{k-2}	b_{k-3}	b_{k-4}	b_k	a_k
0	1	1	1	0	1	1
1	1	1	1	1	0	1
2	0	1	1	1	0	1
3	0	0	1	1	1	0
4	1	0	0	1	0	1
5	0	1	0	0	0	0
6	0	0	1	0	0	0
7	0	0	0	1	1	0
8	1	0	0	0	1	1
9	1	1	0	0	1	0

Table 5-3 State Table for Transforming

Successive States of an FSR to 0100010111

From Table 5-3

$$a_k = f(b_{k-1}, b_{k-2}, b_{k-3}, b_{k-4}) = b_{k-1}\, b'_{k-2} + b_{k-2}\, b_{k-3}$$

The overall cost of an FSR transformation for generating $\{a_k\}$ is

$$4\delta + 7\gamma \text{ (feedback)} + 6\gamma \text{ (transformation)}$$

This cost is lower than that of the (10, 4) BRS generator and equal to that of the (10, 5) BRS when $\delta/\gamma = 13$.

The MIN-6 program is organized to accept a sequence of problems to be solved independently. This flexibility makes it possible to investigate various approaches and assignments in the synthesis of sequential networks.

5.3 Binary Sequence Detector

Binary sequence detectors may be used in ground decoding equipment for locating each successive data frame. An identifier (prescribed sequence) appears at the beginning of each serialized data frame. See subsection 5.2. The sequence detector is analogous to an electronic combination lock which remains closed until a prescribed sequence is entered. It is opened only for the CPI following the last bit in the sequence.

The detector of any given n-bit sequence may be viewed as a sequential network having one input and one output. The sequential network must be capable of assuming at least n-distinct internal states. The minimum number of memory elements required is $1 + \lceil \log_2 n \rceil$ as previously defined. Given the cost of the memory and decision elements, there is no known algorithm for assigning state-values to $1 + \lceil \log_2 n \rceil$ or more memory elements such that the overall cost of the sequential network is minimized. Exhaustive comparisons of state assignments are beyond the reach of present-day general-purpose computers except for minimum state networks where n is less than 9. The binary sequence detector represents a very special class of sequential networks and may therefore be treated accordingly.

The familiar shift register together with an n-input decision element can serve to detect any given n-bit sequence. The register serially stores n-1 bits. These and the $n^{\underline{th}}$ bit (just prior to entering the register) are sensed by an n-input decision element. Thus the given n-bit sequence can be located wherever it occurs. Though straightforward, this method is uneconomical in terms of the number of memory elements required. For large n, the number of decision elements to

practically realize an effective n-input decision element is also significant.

When the given sequence is subperiod free, an alternate approach can be used. The steps in the synthesis procedure [P68-1] are illustrated in Example 5.4.

EXAMPLE 5.4

Given the following sequence

1 1 0 1 1 1 0 0 0 0 $\{a_k\}$

9 8 7 6 5 4 3 2 1 0 k

The sequence $\{a_k\}$ has an (n, r_{min}) BRS representation of (10, 4). This is a minimum r_{min}. The ten 4-bit subsequences are tabulated in Table 5.4.

k	a_{k-1}	a_{k-2}	a_{k-3}	a_{k-4}	a_k
0	1	1	0	1	0
1	0	1	1	0	0
2	0	0	1	1	0
3	0	0	0	1	0
4	0	0	0	0	1
5	1	0	0	0	1
6	1	1	0	0	1
7	1	1	1	0	0
8	0	1	1	1	1
9	1	0	1	1	1

TABLE 5.4 (10, 4) BRS GENERATOR OF 1 1 0 1 1 1 0 0 0 0

An FSR can be used to realize the (10, 4) BRS generator with the following feedback function

$$a_k = a_{k-1}\, a'_{k-2} + a'_{k-3}\, a'_{k-4} + a_{k-2}\, a_{k-3}\, a_{k-4}$$

The six unspecified states are treated as don't cares. Thus

$$a'_{k-1}\, a_{k-2}\, a'_{k-3} + a'_{k-2}\, a_{k-3}\, a'_{k-4} + a_{k-1}\, a_{k-2}\, a_{k-3}\, a_{k-4}$$

$$+ a_{k-1}\, a'_{k-2}\, a'_{k-3}\, a_{k-4} = 0$$

The (10, 4) BRS characterization of $\{a_k\}$ and the associated FSR implementation suggest an organization of a sequential network for detecting $\{a_k\}$ within serialized binary data. In Table 5-5 the internal states of the proposed sequential network are labeled numerically with an initial state designation of 1. The number of internal states, 10, is the number of bits in the sequence. The input to the detector is represented by the Boolean variable x.

Present State	Next State		Present Output	
	x = 0	x = 1	x = 0	x = 1
1	(2)	1	0	0
2	(3)	1	0	0
3	(4)	1	0	0
4	(5)	1	0	0
5	5	(6)	0	0
6	2	(7)	0	0
7	2	(8)	0	0
8	(9)	1	0	0
9	3	(10)	0	0
10	2	(1)	0	1

TABLE 5-5 STATE TABLE FOR A BINARY SEQUENCE DETECTOR

The arrival of the first 0 on the x input line (i.e., possible start of the sequence, a_0) causes the state transition from 1 (initial present state) to 2 (next state). Should each succeeding bit be part

of the sequence to be detected, the sequential network progresses through each state in numerical order. This is indicated in the encircled next states in the state table. During the time the network is in the present state 10 and 1 is on the line, the detector's (present) output is 1. The output is a function of the total state, input state and internal state, of the network. This is a Mealy model of the sequential network.

If at any time a bit is received which is not in the sequence, though previous bits were identical to the start of the sequence, the network must return to the initial state 1, state 2 or 3, or remain in state 5. Since the sequence begins with a 0, whenever a 1 arrives improperly located in the sequence, the network must return to state 2 or 3 or remain in state 5 if preceded by a run of four 0's. For example, assume the network is in present state 9 (meaning the 8 previous bits correspond to the first 8 bits in the sequence) and the $9^{\underline{th}}$ bit is a 0 instead of a 1. Clearly the network should not progress to the state 10. It should instead return to state 3 since bit 8 and bit 9 (now entering) correspond to the first two bits in the sequence. Thus the $8^{\underline{th}}$ bit of the 9-bit block could possibly be the start of the sequence.

It is proposed that the state assignment be taken from the ordered subsequence in the (n, r_{min}) BRS such that:

1) Successive states through which the detector progresses when the sequence is entered are made to correspond to successive subsequences, and
2) An initial state is chosen whereby one of a total of r_{min} delay units in the detector will track the input x at all times.

A state assignment satisfying steps 1 and 2 appears in Table 5-6. Four delay units are required for the detector in Example 5.4. Let $d_1\ d_2\ d_3\ d_4$ and $D_1\ D_2\ D_3\ D_4$ represent the present and next internal state, respectively.

Table 5-6 is divided into three parts for explanatory purposes. The top 10 entries describe the detector's behavior when $\{a_k\}$ is entered. The next 10 entries correspond to a present to next state transition when the input x is not properly in $\{a_k\}$. A total state $x\ d_1\ d_2\ d_3\ d_4$ of 0 1 1 0 1 indicates that the 8 bits previously entered correspond to the first 8 bits in $\{a_k\}$. The present input x is 0 whereas the $9^{\underline{th}}$ bit of $\{a_k\}$ is 1. The next state $D_1\ D_2\ D_3\ D_4$ is 0 0 0 0 or state 3 since the previous and present input could be the start of $\{a_k\}$. The lower portion of Table 5-6 contains unspecified (i.e., ususued) total states. The next internal states are therefore treated as don't cares.

The next state of each delay unit and the present output of the detector, denoted as Z, are Boolean functions of x, d_1, d_2, d_3, and d_4. D_1, D_2, D_3, D_4 and Z may be expressed in (disjunctive or conjunctive) canonical form directly from Table 5-6. These functions represent a multioutput combinational logic network.

Four of the outputs serve as inputs to the delay units. MIN-6 was used in the simultaneous minimization of the next-state functions. Only one specified canonical input (i.e., total state) is associated with Z. In an effort to reduce the number of simultaneous functions in the computer minimization, Z is treated as a single output function.

x	d_1	d_2	d_3	d_4	Present (internal) state	D_1	D_2	D_3	D_4	Next (internal) state
0	0	0	1	1	1	0	0	0	1	2
0	0	0	0	1	2	0	0	0	0	3
0	0	0	0	0	3	1	0	0	0	4
0	1	0	0	0	4	1	1	0	0	5
1	1	1	0	0	5	1	1	1	0	6
1	1	1	1	0	6	0	1	1	1	7
1	0	1	1	1	7	1	0	1	1	8
0	1	0	1	1	8	1	1	0	1	9
1	1	1	0	1	9	0	1	1	0	10
1	0	1	1	0	10	0	0	1	1	1
1	0	0	1	1	1	0	0	1	1	1
1	0	0	0	1	2	0	0	1	1	1
1	0	0	0	0	3	0	0	1	1	1
1	1	0	0	0	4	0	0	1	1	1
0	1	1	0	0	5	1	1	0	0	5
0	1	1	1	0	6	0	0	0	1	2
0	0	1	1	1	7	0	0	0	1	2
1	1	0	1	1	8	0	0	1	1	1
0	1	1	0	1	9	0	0	0	0	3
0	0	1	1	0	10	0	0	0	1	2
0	0	0	1	0		ø	ø	ø	ø	
0	0	1	0	0		ø	ø	ø	ø	
0	0	1	0	1		ø	ø	ø	ø	
0	1	0	0	1		ø	ø	ø	ø	
0	1	0	1	0		ø	ø	ø	ø	
0	1	1	1	1		ø	ø	ø	ø	
1	0	0	1	0		ø	ø	ø	ø	
1	0	1	0	0		ø	ø	ø	ø	
1	0	1	0	1		ø	ø	ø	ø	
1	1	0	0	1		ø	ø	ø	ø	
1	0	1	0	1		ø	ø	ø	ø	
1	1	1	1	1		ø	ø	ø	ø	

TABLE 5-6 STATE TABLE FOR A BINARY SEQUENCE DETECTOR

The simultaneous minimized solutions of D_1, D_2, D_3, and D_4 together with a minimized Z represent a two-level AND-OR diode matrix implementation of the detector's combinational logic.

For Example 5.4

$$D_1 = x\, d_2\, d_3\, d_4 + \underline{d_2\, d_3'\, d_4'} + \underline{x'\, d_1\, d_2'} + x'\, d_3'\, d_4'$$

$$D_2 = \underline{d_2\, d_3'\, d_4'} + \underline{x'\, d_1\, d_2'} + x\, d_1\, d_2$$

$$D_3 = x$$

$$D_4 = x\, d_2' + d_3$$

and $$Z = x\, d_1'\, d_3\, d_4'$$

The cost of the detector is 4 memory elements and 31 diodes. The number of diodes required without minimization is 140! (See Table 5-6). In practice an inverter is required to generate x'. However, signal conditioning of x would be needed for any detection method. It will be assumed that the signal conditioner will provide the assertion and negation of x.

The cost of the multioutput combinational logic for the implementation of D_1, D_2, D_3, and D_4 in Example 5-4 was determined for each possible initial state. These costs appear in Table 5-7. Initial state assignments for which one of the delay units tracks the input, x, results in a lower diode cost than the remaining choices. In particular, the initial state 0 0 1 1 for which $D_3 = x$ yields a minimum cost.

Initial State	Diode Cost	
1 1 0 1	32	$D_1 = x$
0 1 1 0	31	$D_2 = x$
0 0 1 1	27	$D_3 = x$
0 0 0 1	35	$D_4 = x$
0 0 0 0	45	
1 0 0 0	66	
1 1 0 0	67	
1 1 1 0	64	
0 1 1 1	56	
1 0 1 1	54	

TABLE 5-7 DIODE COST VERSUS INITIAL STATE FOR DETECTOR IN EXAMPLE 5-4.

5.4 Digital Techniques For Generating a Time Dependent Acceleration Voltage For a Mass Spectrometer

5.4.1 Introduction

An unmanned interplanetary flight to Mars has been proposed for 1971. An entry probe is to be released from the spacecraft for a descent into the Martian atmosphere. The determination of the constituents of the Martian atmosphere and their relative abundance is one of the scientific goals.

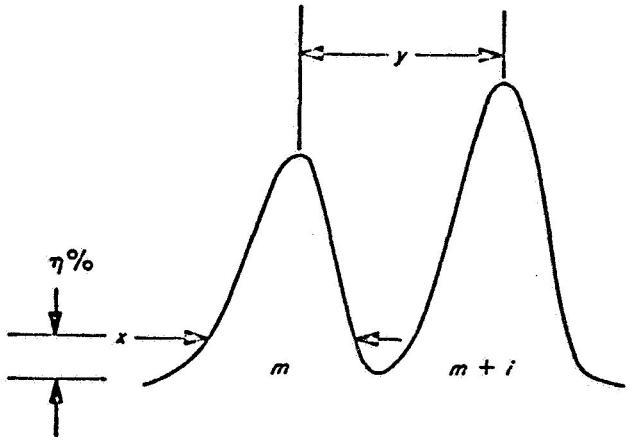
A single focusing mass spectrometer [D58] was first considered. The essential components of the instrument appear in Fig. 5-6. The instrument portion is shown in its mechanical configuration whereas the support electronics is represented in functional blocks.

5.4.2 Instrument Operation

The gas to be analyzed is introduced into the ionization chamber. A portion of the sample gas is ionized when bombarded by an electron beam which is parallel to the source exit slit. The high voltage sweep produces an electrostatic field which accelerates the ions through the source exit slit with approximately homogeneous energy. The resulting ion beam is deflected by the electromagnetic field of the analyzer (permanent) magnet such that at a given value of v (high voltage sweep) all ions with a particular mass per unit charge are focused on the collector defining slit. The ion current is collected and fed into a very sensitive operational amplifier called an electrometer. Automatic scale switching provides an increase in dynamic range.

A monotonically varying v is used to separate ions with different masses per unit charge. A plot of the ion current versus time (resulting from a monotonically varying v) yields a spectrogram. The location of a peak in time identifies the associated mass per unit charge and the amplitude of the peak is a function of its relative abundance.

An important parameter is the instrument's resolution. The mass per unit charge, $\frac{M}{q}$, is in atomic mass units where the isotope $^{16}_{8}O$ is taken to be 16. It differs slightly from the chemical scale of atomic weights [L59]. Hereafter, the atomic mass units (a m u) will be referred to as mass (m). The resolution of the instrument is defined at a particular m as follows:



$$\left.\frac{m}{\Delta m}\right)_{\eta\%} = \frac{m}{(m+i)-m} = \bar{m}\,\frac{y}{x} \times 100\%$$

where $\bar{m} = \frac{m + (m+i)}{2}$

and x and y are time measurements. The resolution of the instrument described in this report is:

$$\left.\frac{m}{\Delta m}\right)_{1\%} = 25$$

That is, at mass 25, the instrument has unit resolution.

5.4.3 Parameters For Determining the Acceleration Voltage Curve

A. Ion Ballistics

The ion ballistics of the instrument in Fig. 1 is expressed as follows:

$$R = \frac{144}{B}\sqrt{\left(\frac{M}{q}\right) v}$$

where $R = 3.81$ cm.

$B = 3{,}780$ Gauss

$\frac{M}{q} = m$ is in a m u.

and v is in volts.

Thus,

$$m(t)\, v(t) = 10{,}000$$

At time t the velocity (which is proportional to v) and the mass m of the ions determine its radius of deflection which must be 3.81 cm. to be focused on the collector defining stit. An accelerating voltage which decays exponentially can be approximated by an RC discharge. The base width of the ion peaks over the entire mass range are nearly the same for the exponential accelerating voltage where

$$v(t) = v(0)e^{\frac{-t}{\gamma}}$$

Unfortunately, ion peaks will not appear linearly separated in time.

A linear separation of ion peaks with respect to time is desirable when interpreting a spectogram. The form required for m(t) is

$$m(t) = at + m(0).$$

Thus

$$v(t) = \frac{10{,}000}{at + m(0)}$$

A hyperbolic (i.e., inverse) acceleration voltage cannot be generated by analog methods as readily as the exponential.

Unlike the exponential case, the base width of the ion peaks varies directly with atomic mass unit interval.

B. Mass Range

The mass range for the instrument in question is 10 to 45. Thus v(t) must vary from 1000 to 222.22 volts. A lower limit of 220 volts is actually used. This places the ion peak associated with mass 45 within the spectrum.

5.4.4 Hyperbolic Curve Generation Using Digital Techniques

A. The Derivation of Successive Decremented DC Voltage Levels of Fixed Duration.

The calculus of finite differences [H3] yields the following discrete relationships:

$$m[t(k)] = at(k) + m(0) = at(k) + 10 = m(k)$$

$$v[t(k)] = \frac{1000}{a't(k) + 1} = \frac{1000}{a'k + 1} = v(k) \qquad (5.5)$$

$t(k) = k$ for $k = 0,1, \ldots, 2^r-1$

and r is an integer.

From (5.5) where $v(2^r-1) = 220$ volts,

$$a' = \frac{39}{11(2^r-1)} = \frac{\hat{a}}{2^r-1} .$$

The quantization required for v in quanta is:

$$R = \left| \frac{v(0)}{\Delta v(2^r-2)} \right| = \frac{\left[2^r (\hat{a} + 1) - (2\hat{a} + 1)\right]\left[\hat{a} + 1\right]}{\hat{a}}$$

$\Delta v(k)$ is the forward difference and $\Delta v(2^r-2) = v(2^r-1) - v(2^r-2)$.

Note that $\Delta v(2^r-2)$ is smallest change v undergoes.

$$\frac{\text{Voltage Quantization}}{\text{Time Quantization}} = \frac{R}{2^r} = \frac{\left[\hat{a} + 1 - \left(\frac{2\hat{a} + 1}{2^r}\right)\right]\left[\hat{a} + 1\right]}{\hat{a}}$$

$$\frac{R}{2^r} = \frac{(\hat{a} + 1)^2}{\hat{a}} = 5.8 \text{ for } r \geq 5$$

Thus if time is quantized with r bits ($r \geq 5$), voltage must be quantized to r + 3 bits to recognize $\Delta v(2^r-2)$.

Fig. 5-7 illustrates this method. Time is quantized by means of feedback shift register (FSR) operating synchronously with a constant clock frequency. The nine stage FSR is cycled through 512 internal states. The assertion outputs of the nine stages represent a 9-bit non-weighted code. A two-level diode AND-OR matrix with twelve outputs translates the 9-bit non-weighted to a 12-bit weighted (positional) code. The 12-bit representation is converted to a DC voltage level which is proportional to the magnitude of 12-bit binary number. This is the function of the digital to analog converter. The 1000 to 220 volt hyperbolic sweep appears at the output of the high voltage operational amplifier. Successive decremented levels of a fixed duration appear at the output of the D/A converter.

The number of diodes in the AND-OR matrix which represent the 9 input twelve output truth table in disjunctive canonical form is 4608 for ANDing and 3054 for ORing or a total of 7662 diodes. A silicon on sapphire microelectronic implementation of the diode AND-OR matrix is currently under test.

MIN-6 was used to find a cover of approximate minimum cost. A reduction of 738 diodes or 9.6% was realized in 4 hours and 12 minutes of computer running time. This program was the only one found which could handle the 12 Boolean functions of 9 variables. It has since been improved particularly for the cover options of approximately minimum cost. Further runs will be made with the improved program.

EXAMPLE 5-5 Hyperbolic Curve Generation with 2^5 quanta

Since time is quantized with r = 5 bits, 8 bits are required to recognize $\Delta v(30)$.

$$v(k) = \frac{255}{\frac{39}{11} \cdot \frac{k}{31} + 1} \quad \text{for } k = 0,1, \ldots, 31$$

The largest 8 bit binary number, 255, is used to represent 1000 volts. The feedback function for the 5-stage FSR is

$$a_k = a_{k-3} \oplus a_{k-5} \oplus a'_{k-1}\, a'_{k-2}\, a'_{k-3}\, a'_{k-4}\, a_{k-5}$$

Successive inputs and outputs of a 5 x 8 matrix appears in TABLE 5-8. Note that a_{k-i} has been replaced by x_i. A plot of $Z = Z_1\, Z_2 \ldots Z_8$ in decimal versus k appears in Fig. 5-8.

The 8 Boolean functions of 5 variables were minimized simultaneously under a cover option of approximate minimum cost. In TABLE 5-8 10000 is the initial state and the singular state 00000 is the terminal state which remains until the first stage is set (i.e., x_1

is made a 1). This initial state yielded the best minimum cover of all the possible 32 initial states. The effect of using a different initial state is to cyclically permute the input states relative to the fixed output states. A total of 293 diodes is associated with each of 32 canonical truth tables. A reduction of 119 diodes or 40.6% was realized with 1000 as an initial state. The initial state of 10101 yielded the smallest reduction of 67 diodes or 22.8%. Each of the minimization runs required less than 2 minutes of IBM 7094 computing time. This included pre-processing, extraction, and post-processing time.

An alternate approach is discussed in [P68-2] whereby the duration of successive DC voltage levels is varied such that a hyperbolic curve results with equal changes in voltage levels.

REFERENCES

[M54] Muller, D. E., Application of Boolean Algebra to Switching Circuit Design to Error Detection, IRE Transactions to Electronic Computers, pp. 6-12, 1954.

[D58] Duckworth, H. E., Mass Spectroscopy, Cambridge University Press, New York, 1958.

[R58] Roth, J. P., Trans. of the American Mathematical Society, vol. 88, pp. 301-326 (1958).

[L59] Leighton, R. B., Principles of Modern Physics, McGraw-Hill Book Company, Inc., New York, 1959.

[R59] Roth, J. P., Algebraic Topological Methods in Synthesis, Proc. of an International Symposium on the Theory of Switching, April 2-5, 1957.

[ERW] Ewing, A. C., Roth, J. P., and Wagner, E. G., Algorithms for Logical Design, AIEE Transactions, Feb. 1961.

[H62] Hamming, R. W., Numerical Methods for Scientists and Engineers, McGraw-Hill Book Company, Inc., New York, 1962.

[Y62] Yoeli, M., Binary Ring Sequences, American Mathematical Monthly, vol. 69, pp. 852-855, November 1962.

[R65] Roth, J. P., Systematic Design of Automata, American Federation of Information Processings Society Conference Proceedings, 27 Part I Fall Joint Computer Conference, Spartan Books, Washington D. C., MacMillan and Co., Ltd., London 1965, pp. 1093-1100.

[M66] Miller, R. E., Switching Theory Vol. I: Combinational Circuits, John Wiley and Sons, Inc., New York 1965.

[G67] Golomb, S. W., Shift Register Sequences, Holden-Day, Inc., San Francisco, 1967.

REFERENCES (Cont'd)

[P68] Perlman, M., The Classification of (n, r) Binary Ring Sequences, Space Programs Summary 37-50, vol. III, pp. 222-225, Jet Propulsion Laboratory, Pasadena, California, April 30, 1968.

[P68-1] Perlman, M., The Synthesis of Binary Sequence Detectors, IEEE Transactions on Computers, vol. C-17, No. 9, September 1968. To appear.

[P68-2] Perlman, M., Digital Techniques for Generating a Time Dependent Accelerating Voltage for a Mass Spectrometer, Space Programs Summary 37-51, vol. III, pp. to , Jet Propulsion Laboratory, Pasadena, California, June 30, 1968.

[RW68] Roth, J. P., and Wagner, E. G., A Calculus and an A Algorithm for a Logic Minimization Problem Together with an Algorithmic Notation. To appear.

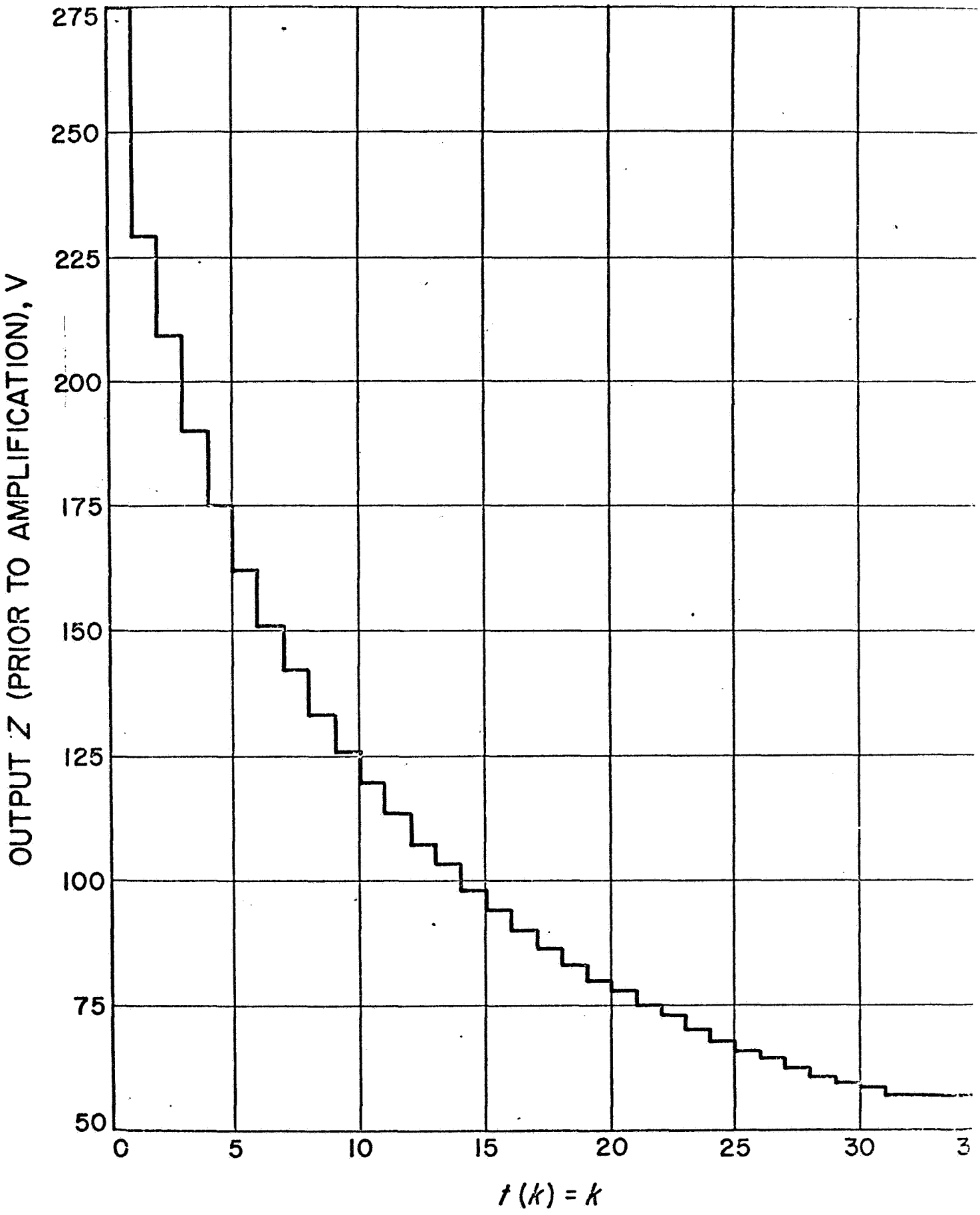


Figure 2.8. Output Z in Decimal Versus t(k) = k.

A Calculus and an Algorithm for a Logic Minimization Problem Together with an Algorithmic Notation

by

J. Paul Roth
E. G. Wagner
Research Division
Yorktown Heights, New York

ABSTRACT: This paper provides a calculus and an algorithm for the 2-level multiple-output minimization problem, an outstanding problem in the theory of switching. In addition a notation, called the F-notation, is introduced for the description of such an algorithm. Two examples are given showing the workings of the algorithm, together with a complete proof for the validity of the method.

This work was supported by NASA's Jet Propulsion Laboratory of the California Institute of Technology, under Contracts 952341 and 951538, and the IBM Corporation.

RC 2280 (#11211)
November 12, 1968

SUMMARY

A calculus and an algorithm for the 2-level multiple-output logic-minimization problem, an outstanding problem in switching theory, is given. (The problem may also be described as that of finding a realization in two levels of any mapping of strings of binary digits into strings of binary digits; the problem in logic of minimum normal form is the special case for mapping into one bit). This problem has a continuing application in the design of computer hardware [RP] currently notably for LSI.

Although algorithms for the single output problem have been known and used for more than a decade, there has been no entirely satisfactory treatment of the multiple-output problem.

D. E. Muller [Mu] proposed a method of encoding, transforming a multiple-output problem into a single-input problem and this method was implemented in a program MIN6[ERW, RP] which has been used extensively for multiple-output problems; this technique does not, in general, obtain a minimum. Various "tag" methods have also been proposed. We do not know whether these methods obtain a minimum but at any rate no proof has been given that in fact they do.

Bartee's method [B], for example, was programmed and applied to practical logic circuit design. Quine's work for single output is classical [Q].

Here the notion of singular cubes is adapted as a natural means for description of the problem and a calculus of singular cubes is provided for carrying out the various necessary computations. Based upon this notation we give an algorithm, a generalization of the (single-output) extraction algorithm. We prove in rigorous fashion that this algorithm always computes a minimum. The cost function used here is a strong generalization of the classical cost function, easily handling LSI technologies. Appendices 1 and 2 are detailed examples of manual execution of the program.

One of the difficulties of the Muller encoding is the expansion of the number of prime cubes (prime implicants) that its usage requires. Appendix 3 gives a simple example which gives a comparison of the number of prime cubes in the approach of this paper with Muller's; our requires 3 prime cubes, his 39.

The algorithm itself is described in a notation, termed the F-notation, a "language" for the specification of algorithms. This notation, which is described here is one of the contributions of the paper. Following the "F-description" here given,

this Multiple Output Minimization algorithm, called MOM, has been programmed by Leon Levy and run in the APL notation [FI] on the IBM APL\360 interpretive system, on the model 50, to be reported elsewhere [RWL].

---

1. Preliminaries

In the main body of this paper we state and solve the multiple-output two-level minimization problem within a strictly mathematical framework. In this section we will state the problem informally in terms of logical circuits and, at the same time, present the notation for logical functions and circuits which underlies the mathematics of the following sections.

A two-level single-output logic circuit consisting of a level of ANDs followed by an OR circuit may be used to implement (or realize) any Boolean function. Such a logic circuit corresponds directly to a normal form expression for the function. However when we consider such circuits with several outputs, that is, circuits where we have a level of ANDs followed by a level of several OR circuits then neither the function realized nor the structure of the circuit can be conveniently represented by a Boolean expression . While the function can be represented by several Boolean functions, trouble arises in representing the structure when, as in Figure 1, one AND block drives more than one output OR since in this event the logical term corresponding to this AND would be repeated in the different expressions despite the fact that it corresponds to a single element in the circuit.

How then can we represent a multiple-output two-level circuit and/or the function which it is supposed to realize? A simple, direct and, as we shall show, fruitful representation is the following "tag" notation. Let us first consider the representation of a circuit. Say the circuit has input lines labelled $a_1, a_1, \ldots, a_r$ and output line labels $b_1, b_2, \ldots, b_s$. Then any given AND block C in the circuit can be conveniently represented by an 'object'

$$a_1\ a_2 \ldots a_r \mid b_1\ b_2 \ldots b_s$$
$$\alpha_1\ \alpha_2 \ldots \alpha_r \mid \beta_1\ \beta_2 \ldots \beta_s$$

where $\alpha_k = 1$ if a_k is a non-negated input to C, $\alpha_k = 0$ if a_k is a negated input to C, and $\alpha_k = X$ otherwise; and $\beta_k = 1$ if C feeds output b_k, $\beta_k = X$ otherwise. We call such an 'object' a singular cube. The left side $a_1, \ldots, a_r$ of a singular cube is known as the input part, the right side, $b_1, \ldots, b_s$ is known as the output part. The circuit-as-a-whole is then represented by a set (list, table) of such singular cubes, one for each AND in the circuit. We call such a set (list, table) a cubical cover or, simply, cover. An example of a cover, and the block diagram of the corresponding circuit, is shown in Figure 1.

In section 2 and in much of section 3 we shall actually employ a somewhat more general notion of a singular cube in that we shall permit 0's as well as 1's and X's to appear in the output parts of singular cubes. This more general notion of singular cube is not relevant to the multiple-output two-level problem but is important in more general forms of circuit decomposition and synthesis (see [R-2]). We employ the more general notion in sections 2 and 3 only in order to make the definitions consistent with those in future papers; also, the increased generality adds little complexity to the definitions. In order to distinguish the two cases we refer to a singular cube

	a	b	c	d	e	f	g	h
Care Conditions	1	1	x	x	x	x	1	x
	x	x	1	0	x	x	1	1
	x	x	1	1	0	0	x	1
Don't Care Conditions	x	x	x	0	0	0	x	x
	x	x	0	x	0	0	x	x

Singular Cube Table

a	b	c	d	e	f	g	h
1	1	x	x	x	x	1	x
x	x	1	0	x	x	1	1
x	x	x	x	0	0	x	1

A "Cubical Cover"

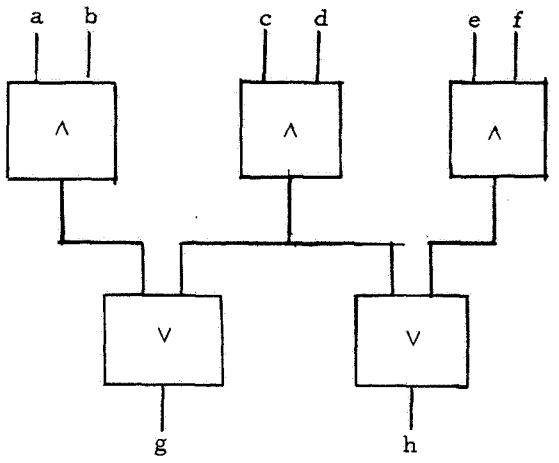


Figure 1 Example of the 2-level Minimization Problem

in which there are no 0's in the output part as an on-cube.

The above representation of a circuit not only tells us the inputs of the ANDs and which outputs they are connected to, it also represents the function realized by the circuit. Briefly, any combination of input signals (i.e., any r-tuple of 1's and 0's) which results from replacing the X's in the input part of a singular cube with 1's or 0's will produce 1's on those outputs corresponding to 1's in the output part of the singular cube and the output will be unspecified for those outputs corresponding to X's in the output part of the singular cube. If, for a given cubical cover, combination of input signals, and output b_j, no singular cube specifies an output of 1 on b_j for this input then this output will be a 0. Given a circuit C represented by a cubical cover we say it realizes the function (from r-tuples of 1's and 0's to s-tuples of 1's and 0's) given by the above rules.

In general any set of on-cubes (with r input and s outputs) will correspond to a possibly incomplete specification of some switching function mapping r-tuples of 1's and 0's into s-tuples of 1's and 0's. (The situation where 0's are allowed in the output parts of the cubes is somewhat more complex.)

In practice, that is, in the design of practical circuits, there are many situations in which it is not necessary to specify what the output signals should be for every combination of input signals. Sometimes this fact can be taken advantage of to produce a more economical circuit by judicious choice of output signals for these DON'T-CARE combinations of input signals. In order to be able to cope with this situation we allow a function to be specified by using two sets (lists, tables) of singular cubes. One set of singular cubes specifies the CARE-CONDITIONS the other specifies the DON'T-CARE-CONDITIONS. If a singular cube appears in the CARE-CONDITIONS this means that we want the outputs corresponding to the 1's in its output part to be on for any combination of input signals that results from replacing the X's in its input part with 1's and 0's. If a cube is in the DON'T-CARE-CONDITIONS then this means that we "don't care" (i.e., we are not specifying) what the outputs should be for any combination of input signals that results from replacing the X's in its input part with 1's and 0's. An example of a function with DON'T CARES is shown in Figure 1.

Given a switching circuit C realizing some function $F: \{0,1\}^r \rightarrow \{0,1\}^s$ and a switching function G with DON'T-CARES given by means of sets of singular cubes, we say that C is an implementation of G if the functions F and G agree on the combinations of input signals specified by the CARE-CONDITIONS of G.

Now in general there is some cost function associated with a logical circuit. This cost may be, for example, proportional to the number of ANDs, or proportional to the number of inputs to the ANDs, or proportional to a linear combination of the number of inputs to the ANDs and the number of inputs to the ORs. Thus one is in general not only interested in implementing a given switching function G, but in implementing it at the lowest possible cost. What we do in the remaining sections of this paper is develop an algorithm which, for a great variety of cost functions, will, given any switching function G (with or without DON'T-CARES) find a circuit (cubical cover) which implements G and which is of minimum cost with respect to the given cost function.

2. Singular-Cubes, - Complexes and - Covers

In the example of Figure 1, the array describing the function to be implemented contains the row

a b c d e f g h
1 1 x x x x | 1 x

Here an x on the left has the interpretation that the associated "variable" or "label"—here c, d, e or f- may have the value 1 or 0 i.e., values may be arbitrarily assigned and still the corresponding functional value is unchanged. The x assigned to the output variable h is to be thought of as a value unspecified by this cube. The entire object

a b c d e f g h
1 1 x x x x | 1 x

is termed a singular cube.

In general, given a function having r inputs and s outputs, its functional behavior will be described by an ensemble or, more specifically, by what we term a cover C of singular cubes called the CARE conditions and a cover D of the DON'T CARE conditions (in general an implementation commits the DON'T CARE conditions); a singular cube will then be a pair consisting of an r-tuple $a_1, \ldots, a_r$ of symbols 0, 1 or x

called the "input part" and an s-tuple $(b_1,\ldots,b_s)$ of such symbols, $b_j = 0, 1$ or x, called the "output part" written in the form $a_1,\ldots,a_r \mid b_1,\ldots,b_s$. In general if such a singular cube belongs to a cover C defining a function f, the x's on the left have the meaning that they may be changed arbitrarily to 1's and 0's to describe the behavior of the function; an x on the right indicates that this particular cube does not specify a value for the function for this particular output coordinate.

Does any set of singular cubes having the same input coordinates and output coordinates define a function? The answer is no, they must satisfy a certain consistency condition to insure that the function is single-valued. In order to define this precisely it is convenient to first define interface of cubes and singular cubes, and the notion of a complex.

The <u>interface</u> of two cubes corresponds to forming the AND of the corresponding logical terms. Let $a_1,\ldots,a_n$ and $b_1,\ldots,b_n$ be cubes, with a_i and b_j equal to 1, 0 or x. The <u>interface</u> $\sqcap$ is defined with the aid of the following table

$\sqcap$	0	1	x
0	0	$\overline{0}$	0
1	$\overline{0}$	1	1
x	0	1	x

where the symbol $\overline{0}$ stands for the <u>null cube</u>. Then if $a_i \sqcap b_i = \overline{0}$ for any i then $a \sqcap b = \overline{0}$. Otherwise $a \sqcap b = a_1 \sqcap b_1 \ldots a_n \sqcap b_n$. Let then $a \mid b$ and $c \mid d$ be singular cubes with a c, and b and d having the same coordinate labels. Then we define

$$(a \mid b) \sqcap (c \mid d) = (a \sqcap c) \mid (b \sqcap d).$$

For example, $(0\ 1\ x \mid 1\ x) \sqcap (1\ x\ 1 \mid x\ 0) = \overline{0}$

whereas $(0\ 1\ x \mid 1\ x) \sqcap (x\ 1\ 0 \mid 1\ 0) = 0\ 1\ 0 \mid 1\ 0.$

It is first necessary to introduce some operations as part of the definition of the singular complex. The first of these is the face operator. The i^{th} (input) <u>face-operator</u> ∂_i^α where i is an input label and $\alpha = 0$ or 1 is defined by the expression

$$\partial_i^\alpha (a \mid b) = (x\ldots x, \overset{i}{\alpha}, x\ldots x \mid x\ldots x) \mid \mid (a \mid b)$$

This has the effect of changing the i^{th} input coordinate to an α if this coordinate is an x; when this coordinate is α, then no change is made; when it is $\overline{\alpha}$, the result is $\overline{0}$, the null cube.

Thus for example:

$$\partial_2^1(1\ x\ 1\ x \mid 1\ 0) = 1\ 1\ 1\ x \mid 1\ 0$$

$$\begin{matrix} 1\ 2\ 3\ 4 & 1'2' & 1\ 2\ 3\ 4 & 1'2' \end{matrix}$$

If j is an output label and $b_j \neq x$ then the (output) coface operator δ_j is defined as the cube obtained from $a \mid b$ by replacing b_j by x. If $b_j = x$ then δ_j is the identity mapping. For example:

$$\delta_{2'}(1\ x\ 0\ 0 \mid 0\ 1\ x) = 1\ x\ 0\ 0 \mid 0\ x\ x$$

$$\begin{matrix} 1\ 2\ 3\ 4 & 1'2'3' & 1\ 2\ 3\ 4 & 1'2'3' \end{matrix}$$

A cube $a \mid b$ is said to be the face of the cube $c \mid d$ if there exist input-face operators $\partial_i, \partial_j, \ldots, \partial_k$ and output-coface operators $\delta_q, \delta_v, \ldots, \delta_s$ such that

$$\partial_i \partial_j \ldots \partial_k \delta_q \delta_v \ldots (c \mid d) = (a \mid b).$$

If $a \mid b$ is a face of $c \mid d$ we denote this by writing $(a \mid b)$

$$(a \mid b) \sqsubset (c \mid d).$$

A singular cube of the form

$$v_1 v_2 \ldots v_n \mid X \ldots X\ \alpha\ X \ldots X$$

where $v_i = 0$ or 1, $\alpha = 0$ or 1 is called a (singular) vertex. Clearly, to each singular cube $a \mid b$ there corresponds a unique set of singular vertices which are faces of $a \mid b$.

A singular complex K is an ensemble K of singular cubes with the following properties:

1) For $a \mid b$ and $c \mid d$ singular cubes of K, then $a \sqcap c \neq \bar{0}$ implies $b \sqcap d \neq \bar{0}$. This is called the consistency condition.
2) If $a \mid b$ is a singular cube of K then so are all its faces.
3) If all the singular vertices of a singular cube $a \mid b$ are in K then $a \mid b$ is also in K.

By virtue of the consistency condition each singular complex corresponds to a function mapping strings of binary digits into strings of binary digits.

A singular cube z of complex K is said to be a prime cube if it is not the face of any other cube of K (for the single output case with no DON'T-CARE conditions, prime cubes correspond to prime implicants).

We shall say that complex L is a subcomplex of complex K if every cube of L is also a cube of K, denoted $L \sqsubset K$ or $K \sqsupset L$.

Thus, for example, if K is a complex, the ensemble L of cubes formed by taking the coface with respect to all but the first output coordinate would be a subcomplex—it might be

termed a single-output subcomplex of K.

A subset C of a complex is called a cover. If L is a complex and $C \subset L$ is a cover then let K(C) denote the complex determined by C, i.e., K(C) is the complex such that

i) every element of C is in K(C)

ii) if $u \in K(C)$ then every input-face and output-coface of u is in K(C).

iii) if all the vertices of a singular cube are in K(C) then so is the cube itself.

iv) no cube is in K(C) except those determined by the above rules.

Given a complex K and given S, $C \subset K$ we say that C is an S-cover of (a subcomplex) L (of K) if every vertex of L is a face of some cube of $S \cup C$, (or, equivalently, if L is a subcomplex of $K(S \cup C)$). When $S = \phi$ (the empty set) we shall say C is a cover of L (rather than ϕ-cover).

Given covers C and C' we say that C covers C' if C is a cover of K(C'). We say that two covers C and C' are equipotent, and write $C \tilde{=} C'$ if $K(C) = K(C')$, (note $C \tilde{=} K(C)$).

Let $\equiv$ be the relation on the set of singular cubes (of some complex K) such that $a \mid b \equiv c \mid d$ if, and only if, $a = c$ i.e., iff the two cubes have the same input part. Given a cover $C = \{a_1 \mid b_1, \ldots, a_n \mid b_n\}$ let $\pi(C) = \{P_1, \ldots, P_k\}$ be the partition of C in accordance with $\equiv$. Then we define the reduction of C to be the cover consisting of all cubes $c_y \mid d_y$ such that

$$c_y \mid d_y = \bigsqcap_i (a_i \mid b_i) \quad (a_i \mid b_i \in P_j).$$

The reduction, R(C), of a cover C is thus the cover which results from replacing all cubes with the same input part by their interface. Clearly, for all C, $R(C) \tilde{=} C$. Given two covers C, C' we write $C \approx C'$ if $R(C) = R(C')$.

Just as a complex corresponds to a switching function so does a reduced cover correspond, as indicated in section 1, to a two level AND-OR circuit. If $K(C) \tilde{=} L$ then R(C) corresponds to a circuit realizing L. If K is a complex, L is a subcomplex of K and $C \subset K$ is a cover of L then R(C) corresponds to a circuit realizing L with DON'T-CARE conditions, given by K-L.

In as much as our interest is in producing circuits at minimum cost we must introduce a notion of the cost of a cover. For the purposes of this paper we will introduce a quite general notion of cost which is applicable in many different situations.

To begin with we assume that

A. 1) the cost $c(a\,|\,b)$ of an individual singular cube $a\,|\,b$ is less than that of its input-faces and more than that of its output-cofaces.

The motivation for the following assumption is given by proposition 3 below.

A. 2) Given singular cubes $a\,|\,q$, $a\,|\,r$, $a\,|\,s$ (all with the same input part) then

a) $$c(a\,|\,q \sqcap r) \leq c(a\,|\,q) + c(a\,|\,r)$$

and

b) $$c(a\,|\,q \sqcap r \sqcap s) \leq c(a\,|\,q \sqcap s) + c(a\,|\,r \sqcap s) - c(a\,|\,s)$$

Now, given a cover C we define its cost $c(C)$ to be the sum of the costs of the individual cubes in the reduction, $R(C)$, of C. That is, if $R(C) = \{a_1\,|\,b_1, \ldots, a_n\,|\,b_n\}$ then

$$c(C) = \Sigma_i\ c(a_i\,|\,b_i).$$

Note that these above requirements on the notion of cost are such as to allow the possibility that the cost of adding a cube $a\,|\,b$ (or a cover C') to a cover C may depend on what is already in C.

For example: if we add $a\,|\,b$ to C but C contains a cube $a\,|\,c$ with $a\,|\,b \sqsubset a\,|\,c$ then we see that $R(C \cup \{a\,|\,b\}) = R(C)$ and so $c(C \cup \{a\,|\,b\}) = c(C)$; while, on the other hand, if there is no cube in C with input part a then $R(C \cup \{a\,|\,b\}) = R(C) \cup \{a\,|\,b\}$ and $c(C \cup \{a\,|\,b\}) = c(C) + c(a\,|\,b)$.

Thus the cost c has associated with it an relative-cost function c^* such that for each pair of covers C and C' (drawn from some common complex) we have

$$c^*(C, C') =_{def} c(C \cup C') - c(C)$$

is the cost of adding C' to C.

The following properties of the relative-cost function c^* will be of use. Let K be a complex.

Proposition 2. 1: For all S, C, $C' \subset K$,

$$c^*(S,\ C \cup C') = c^*(S, C) + c^*(S \cup C, C')$$

(Note, for $S = \phi$ take $c^*(S, C) = c(C)$).

Proof: Follows immediately from the definition of c^*.

Proposition 2.2: If $S \subset K$ and $a|b$ and $c|d$ are elements of K with $a \neq c$ then,

$$c^*(S, a|b) = c^*(S \cup \{c|d\}, a|b)$$

Proof: Let $a|e$ be the element of $R(S \cup \{a|b\})$ of which $a|b$ is an output-coface. Clearly since $a \neq c$, $a|e$ is also an element of $R(S \cup \{a|b\} \cup \{c|d\})$. Now, by the definition of cost, if there is no element in R(S) with input part a then $a|e = a|b$ and

$$c^*(S, a|b) = c(a|b) = c^*(S \cup \{c|d\}, a|b);$$

while if there is an element $a|f$ in R(S) then $a|e = (a|b) \sqcap (a|f)$ and

$$c^*(S, a|b) = c(a|e) - c(a|f) = c^*(S \cup \{c|d\}, a|b)$$

since $c|d$ does not affect $a|f$.

Proposition 2.3: If $S \subset K$, a complex, and $a|b, a|d \in K$ then $c^*(S, \{a|b, a|d\}) \leq c^*(S, a|b) + c^*(S, a|d)$.

Proof: Let $a|f$ denote the element, if any, of R(S) with input part a. Then, by the definition d, c and c*

$$c^*(S, \{a|b, a|d\}) = c(a|f \sqcap b \sqcap d) - c(a|f)$$

$$c^*(S, a|b) = c(a|f \sqcap b) - c(a|f)$$

$$c^*(S, a|d) = c(a|f \sqcap d) - c(a|f).$$

But the result then follows from assumption A:2b on cost, and where no element in R(S) has input part a, the result follows from assumption A:2a.

The covering problem then (a generalization of the two-level multiple output minimization problem) takes the following form: Let K be a singular complex containing a subcomplex L; find a cover C of L, where the cubes of C all belong to K, having a minimum cost.

The vertices of L are called the CARE conditions and thus L might be termed the "CARE Complex". The DON'T CARE vertices are in K-L.

We have the following result:

Lemma 2.4: Let C be a cover of complex L by cubes of $K \sqsupset L$, of minimum cost. Then each cube of C is either a prime cube or the output coface of a prime cube.

Proof: Let $a|b$ be an element of C. If it is neither prime nor the output coface of a prime cube of K, then there is a cube $a'|b$ which contains $a|b$ and is itself the face of a prime cube of K. By the definition of containment and cost, $a'|b$ covers all vertices which $a|b$ does and has a lower cost. Thus $(C - (a|b)) \cup (a'|b)$ is also a cover, of lower cost than C, contrary to hypothesis. Q. E. D.

Thus while in the single-output problem, a minimum cover C consists of prime cubes, in the multiple output case, the elements of C may be output cofaces of prime cubes.

3. Elements of the Singular Calculus

The interface (or intersection) of cubes and singular cubes was introduced in section 2. In this section we will introduce the #-product, a kind of differencing operation on cubes. We will show how, when this operation is properly extended to covers, it leads to a convenient algorithm for computing the prime cubes of the complex corresponding to a cover C.

The #-product of cubes is given by the following rules: Let $a|b$ and $c|d$ be singular cubes in some complex K; then

a) If $a \sqcap c = \overline{0}$ then,

$$(a|b) \# (c|d) = (a|b)$$

b) If $a \sqsubset c$ then,

$$(a|b) \# (c|d) = (a|e),$$

where e is such that

$$e_i = \begin{cases} x \text{ if } b_i = d_i \\ x \text{ if } b_i = x \\ b_i \text{ if } d_i = x \end{cases}$$

Note that we cannot have $b_i = 1$, $d_i = 0$ (or vice versa) since $a|b$ and $c|d$ are assumed to be in the same complex.

c) If neither $a \sqcap c = \overline{0}$ nor $a \sqsubseteq c$ then

i) there is one singular cube $a \mid f$ in the result where f is such that

$$f_i = \begin{cases} b_i & \text{if } d_i = X \\ X & \text{otherwise} \end{cases}$$

ii) In addition, for each i such that $a_i = X$, $c_i \neq X$ there is a cube $u \mid b$ such that

$$u_j = \begin{cases} a_j & \text{for } j \neq i \\ 0 & \text{for } j = i,\ c_i = 1 \\ 1 & \text{for } j = i,\ c_i = 0 \end{cases}$$

Note that the set of cubes $u \mid b$ given by ii) is precisely the set of cubes $u \mid b$ such that u is an element of the #-product of cubes a#b as defined in [R-1].

Note that $(a \mid b) \# (a \mid b) = a \mid x$ (by i)).

Since a cube $a \mid x$ specifies no outputs it is, under this interpretation, equivalent to a null cube $\overline{0}$. In what follows we will make this identification.

As an example of the #-product:

	1	x	1	x	1	x	1	x	0	0
#	x	1	1	x	x	1	1	0	x	0
	1	x	1	x	1	x	x	x	0	x
	1	0	1	x	1	x	1	x	0	0

The motivation for calling the #-product a differencing operation is captured by the following result.

Lemma 3.1: $(a \mid b) \# (c \mid d)$ consists of a cover of all faces of $a \mid b$ which are not faces of $c \mid d$.

Proof: a) Suppose that $a \sqcap c = \overline{0}$. Then no face of $a \mid b$ is a face of $c \mid d$, so that $(a \mid b) \# (c \mid d) = (a \mid b)$ indeed consists of all faces of $a \mid b$ which are not a face of $c \mid d$.

b) Suppose that $a \sqsubseteq c$ then the #-product $(a \mid b) \# (c \mid d)$ consists of a single cube $a \mid e$ wherein e has all coordinates x except where $b_i \neq x$ and $d_i = x$: in this case $e_i = b_i$. These correspond precisely to those output coordinates for which $a \mid b$ have faces and $c \mid d$ does not. Hence the conditions of the Lemma hold.

c) In the case that neither a) nor b) hold then the #-product $(a \mid b) \# (c \mid d)$ has two parts. First there are the output coordinates, such as with b) above, where $b_i \neq x$ and $d_i = x$: these are outputs for which $c \mid d$ has no faces. Consequently

the #-product contains a singular cube (a#e) as in the case b) above wherein $e_i = b_i$ except where $b_i = x$ or $b_i = d_i$, in which case $e_i = x$. It is clear that this cube is a face of $a|b$ having nothing in common with $c|d$.

Secondly the #-product a#c is formed: this constitutes a cover of all cubes of a which are not in c. To each such cube f corresponds a singular cube of $(a|b)\#(c|d)$, consisting of $f|b$. Clearly $f \sqcap c = \overline{0}$ so that $f|b$ can have nothing in common with $c|d$.

It now remains to show that all faces of $a|b$ not in $c|d$ are so included. For a face to be in $a|b$ and not in $c|d$ it is necessary either i) that it apply to an output, i.e. have an output coordinate $\neq x$ for which the corresponding output coordinate of $c|d$ is x; or ii) for it to constitute a cube u of the input part $u \sqsubset a$ which is disjoint from c. In the first case, the cube $a|e$ covers all cubes of this type since its input cube is a itself and its output coordinate consists of all x's except precisely where $a|b$ has an output and $c|d$ does not. In case ii) we have formed the set of <u>all</u> singular cubes $f|b$ wherein f is a cube of a#c, so that all cubes of $a|b$ which have input parts disjoint from $c|d$ are so formed. Thus $(a|b)\#(c|d)$ does include all faces of $a|b$ not in $c|d$. Q.E.D.

<u>Lemma 3.2</u> If $(a|b) \sqsubset (c|d)$ and $(a|b) \sqcap e|f) = \overline{0}$ then $((c|d)\#(e|f)) \sqsupset (a|b)$.

<u>Proof:</u> By Lemma 3.1 $(c|d)\#(e|f)$ consists of a cover of all faces of $c|d$ which are not a face of $e|f$. But $a|b$ is such a face.

Q.E.D.

In order to extend the #-product to covers efficiently we introduce the notion of <u>subsuming a cover.</u> Given a cover $C = \{C_1, \ldots, C_n\}$, the corresponding subsumed cover, denoted S(C), is merely the set of elements of C maximal under $\sqsubset$. The process of subsuming a cover is facilitated by means of the following test for $\sqsubset$.

<u>Lemma 3.3:</u> Given singular cubes $a|b$ and $c|d$ then $a|b \sqsubset c|d$ if, and only if,

$$(a|b) \sqcap (c|d) = (a|d),$$

or, equivalently, if

$$(a|b)\#(c|d) = \overline{0}.$$

<u>Proof:</u> Both follow immediately from the relevant definitions.

Given a cover $C = \{a_1|b_1, \ldots, a_n|b_n\}$ we can then find S(C) by forming all interfaces $e_{ij} = (a_i|b_i) \sqcap (a_j|b_j), 1 \leq i \leq j \leq n$ then if $e_{ij} = a_i|b_j$ delete $a_i|b_i$, while if $e_{ij} = a_j|b_i$ then delete $a_j|b_j$. S(C) then consists of all singular cubes from C not

so deleted.

Now, given a cover $C = \{C_1, \ldots, C_n\}$ and a singular cube $a|b$ we define

$$C \# a|b =_{def} S((C_1 \# a|b) \cup \ldots \cup (C_n \# a|b)).$$

Given C as above and another cover $D = \{D_1, \ldots, D_m\}$ we then define

$$C \# D =_{def} S((\ldots((C \# D_1) \# D_2) \ldots \# D_m)$$

$$= S(S(\ldots S(S(C \# D_1) \# D_2) \ldots \# D_m).$$

(That these two definitions are equivalent will follow from the results given below.)

Lemma 3.4: If C and D are covers, and $C \cup D$ is a cover (so that all elements of C and D are elements of some common complex) then $C \# D$ is a cover of those cubes in the complex covered by C which are not in the cover of the complex covered by D.

Proof: Follows easily from Lemma 3.1, the fact that for any cover E, $S(E)$ and E cover the same complex, and from straightforward inspection of the definition of $C \# D$. Q. E. D.

Corollary 3.5: If C and D and $C \cup D$ are covers then

i) C is a cover of D if, and only if $D \# C = \overline{0}$

ii) C and D are equipotent if, and only if,

$C \# D = D \# C = \overline{0}$.

We now turn to the development of the #-algorithm for finding the prime cubes of a cover consisting of on-singular-cubes.

We note first that if C is a cover of on-singular-cubes with input part, say, $i_1, \ldots, i_r$ and output parts $j_1, \ldots, j_s$, clearly then every cube C_k in C is contained in the cube

$$X|1 =_{def} \begin{matrix} i_1\ i_2 \ldots i_r \\ X\ X \ldots X \end{matrix} \Big| \begin{matrix} j_1\ j_2 \ldots j_s \\ 1\ 1 \ldots 1 \end{matrix}.$$

It follows then from Lemma 3.4 that $Z = X|1 \# (X|1 \# C)$ is a cover of the complex covered by C. What we will now do is show that Z consists of precisely the prime cubes of the complex covered by C.

The fundamental lemma is the following:

Lemma 3.6: Let $a|b$ and $c|d$ be on-singular cubes. Then $(a|b) \# (c|d)$ consists of the set of all prime cubes of the complex consisting of the faces of $a|b$ which are not in $c|d$.

Proof: Clearly the result holds when $a|b \# c|d$ is computed by part a) or b) of the the definition of #-product. Thus we consider only the cases using part c), i.e., those in which $a \sqcap c \neq \phi$ and $a \not\sqsubset c$.

Given any cover C let K(C) denote the corresponding complex. By Lemma 3.1 we know that $(a|b)\#(c|d)$ is a cover of $K(a|b) - K(c|d)$; what we must show is that every prime cube $e|f$ of $K(a|b) - K(c|d)$ is in $(a|b)\#(c|d)$.

We proceed by cases according as to whether or not $e = a$.

Case I: Say $e|f$ is a prime cube of $K(a|b) - K(c|d)$ and $e = a$. If for some i, $f_i = d_i = 1$ then we know that

$$a|X\ldots X\overset{i}{1}X\ldots X \in K(e|f)$$

and

$$c|X\ldots X\overset{i}{1}X\ldots X \in K(c|d);$$

but then, since $a \sqcap c \neq \phi$ we have $a \sqcap c \,|\, X\ldots X\overset{i}{1}X\ldots X$ is in both $K(e|f)$ and $K(c|d)$ which contradicts Lemma 3.1. Thus it must be that if $f_i = 1$ then $d_i = X$, but since $e|f \sqsubset a|b$, $f_i = 1$ implies $d_i = 1$, and from this it follows that $e|f$ is contained in the term produced by part i) of rule c in the definition of the #-product. This completes the proof of case I.

Case II: Say $e|f$ is a prime cube of $K(a|b) - K(c|d)$ and $e \neq a$. We claim first that $f = b$ since, one, $e|f \sqsubset a|b$ thus $f_i = 1$ implies $b_i = 1$, and two, if there does not exist i such that $f_i = d_i = 1$ then perforce $a|f \in K(a|b) - K(c|d)$ but then $e|f \sqsubset a|f$ but $e|f \neq a|f$ and this contradicts the primeness of $e|f$, hence there must exist some i such that $f_i = d_i = 1$, but then perforce $e \sqcap c = \overline{0}$ whence $e|b \in K(a|b) - K(c|d)$ but $e|f \sqsubset e|b$ so the primeness of $e|f$ implies $f = b$.

Furthermore since $e|b \sqsubset a|b$ it follows that for all i, $a_i \neq X$ implies $e_i \neq X$.

Now say $e|b$ is not in $(a|b)\#(c|d)$. By definition $(a|b)\#(c|d)$ contains, for each j such that $a_j = X$ and $c_j \neq X$, a cube $t_j = u^j|b$ such that

$$u_i^j = \begin{cases} a_i & \text{for } i \neq j \\ 0 & \text{for } i = j,\ c_j = 1 \\ 1 & \text{for } i = j,\ c_j = 0\,. \end{cases}$$

But then if for all such j, $e|b$ is not contained in $t_j = u^j|b$ it must be the case that for each such j, $e_j = X$ for if for any j, $e_j = u_j^j$ then $e|b \sqsubset t_j$ but if for all j, $e_j \neq u_j^j$ but for one or more j's $e_j \neq x$ then for these j's we have $e_j = c_j$ and hence $e \sqsubset c$. But we know, from the

first part of this case, that there exists i such that $b_i = d_i = 1$, hence $e|X...X^{i}_{1}X...X$ is in both $K(a|b)$ and $K(c|d)$ contradicting Lemma 3.1. Thus $e|b$ must be contained in $t_j \in (a|b)\#(c|d)$ for some j, and so, from the assumption that $e|b$ is prime we get $e|b = t_j$ and thus $e|b \in (a|b)\#(c|d)$ if $e \neq a$. Thus, in any case $e|f \in (a|b)\#(c|d)$. Q. E. D.

Lemma 3.7: If $a|b$ is a singular cube and $C = \{C_1, \ldots, C_n\}$ is a cover then $(a|b)\#C$ is a cover consisting of the prime cubes of the complex defined by these faces of $a|b$ which are not faces of the complex covered by C.

Proof: By definition

$$(a|b)\#C = S(\ldots S(S((a|b)\#C_1)\#C_2)\ldots\#C_n).$$

We proceed by induction on n. For $n = 1$ the result follows from Lemma 3.6. Now for $n > 1$, assuming the result true for $n-1$ say

$$(a|b)\#\{C_1, \ldots, C_{n-1}\} = \{D_1, \ldots, D_m\}.$$

By the induction hypothesis each D_i is a prime cube and every prime cube of the indicated complex is among the D_i. By definition,

$$(a|b)\#\{C_1, \ldots, C_{n-1}, C_n\} = S((D_1\#C_n)\cup\ldots\cup(D_m\#C_n)).$$

Now say there is some prime cube P of $(a|b)\#C$ which is not present in this cover. Since P is in the complex covered by $(a|b)\#C$ it is perforce in the complex covered by $(a|b)\#(C-\{C_n\})$. But then there must exist i such that $P \sqsubset D_i$ since by the induction hypothesis the D_i's are all the prime cubes of the complex covered by $(a|b)\#(C-\{C_n\})$. But then, by Lemma 3.6, P must be an element of $D_i\#C_n$. Thus $(a|b)\#C$ contains all the desired prime cubes. On the other hand, as a consequence of the subsuming operation, $(a|b)\#C$ does not contain any cubes which are not maximal with respect to $\sqsubset$, hence it contains only prime cubes. This completes the inductive step. Q. E. D.

Theorem 3.3: If C is a cover of on-singular-cubes then

$$Z = X|1\ \#\ (X|1\ \#\ C)$$

consists of precisely all prime cubes of the complex covered by C.

Proof: Follows immediately from the above lemma and preceeding remarks. Q. E. D.

4. Fundamentals of the Multiple-Output Extraction Algorithm

The "program" of the multiple-output extraction algorithm in the F-notation is given in section 6. Here we shall describe some of the fundamental aspects of the algorithm informally and present the lemmas which justify them.

Recall, from section 2, that the covering problem was described as follows: We are given a singular complex K containing a subcomplex L and having a cost function c defined on K in accordance with the discussion in section 3. The problem is to find a cover M of L which consists solely of cubes from K and is of minimal cost (i.e., if C' is any other cover of L consisting solely of members of K then $c(C') \geq c(M)$.) The problem as we treat it here is slightly different in that we assume that we start initially from covers C and D such that L (the CARE complex) is the complex corresponding to C and K-L (the DON'T-CARE complex) is the complex corresponding to D.

From Lemma 2.4 we know that the minimal cover M of L must consist of prime cubes or (output) cofaces of prime cubes from K. The #-algorithm presented in section 3 provides a mean for computing the set Z of prime cubes of K, to wit,

$$Z = (X \mid 1\#(X \mid 1\#(C \cup D))).$$

The extraction algorithm operates in an essentially iterative fashion. As will be seen, at any given step in the algorithm we will have sets S, C, D and Z where

S is the partial solution developed so far.

C is a cover of L consisting of prime cubes, or cofaces of prime cubes, of K.

D is a cover of the DON'T CARES.

Z is a set of elements or cofaces of elements from $\ddot{Z}$ (the prime cubes of K) such that $S \cup Z$ is a cover of L, and such that for each $z \in \ddot{Z}$, if $z \notin Z$ then Z contains at most one coface z' of z (i.e. $\ddot{Z}$ is reduced).

The extraction algorithm employs various devices or operations to further develop the solution. The remainder of this section is dedicated to justifying two such operations: the less-than operation and the extremal-finding operation.

Let S, C, D and Z be as described above. Then by an (C, D, S, Z)-cover of L we mean a cover Q of L which consists of S plus elements, or cofaces of elements, of Z. (Such a cover exists since, by definition, $Q = S \cup Z$ is a cover of L.)

The less-than operation is a device for eliminating cubes from Z (given C, D, S, Z); the idea is to eliminate an element of Z if some other element can do "the same job" at lower or equal cost. More precisely, given C, D, S and Z and two elements u and v of Z, we say that u is less-than v, $u < v$, with respect to C, D and S if for every coface u' of u (including the case $u' = u$) there exists a coface v' of v such that $(u' \# (D \cup S)) \# v' = \overline{0}$ and $c(S \cup \{v'\}) \leq c(S \cup \{u'\})$.

Lemma 4.1: Given C, D, S, and Z, as described above, and elements u and v in Z with $u < v$, then there exists a minimum(C, D, S, Z)-cover M of L consisting of elements or cofaces of elements of Z, which does not contain u or any of its cofaces.

Proof: Let Q be any (C, D, S, Z)-cover of L. Say that Q contains cofaces $u_1, \ldots, u_n$ of u. Let $u' = u_1 \sqcap \ldots \sqcap u_n$. From the definitions of cover and cost it is easily seen that Q covers exactly the same complex and has exactly the same cost as the cover

$$(Q - \{u_1, \ldots, u_n\}) \cup \{u'\}.$$

Thus let us assume that $n = 1$ and that u' is the only coface of u in Q (we also wish here to include the case $u' = u$). Since $u < v$ there exists v' (where v' is v or one of its cofaces) such that $(u' \# D) \# v' = \overline{0}$ and $c(S \cup \{v'\}) \leq c(S \cup \{u'\})$. We claim now that $Q^* = (Q - \{u'\}) \cup \{v'\}$ is an (C, D, S, Z)-cover of L such that $c(Q^*) \leq c(Q)$. Since Q is an arbitrary (S, C, D, Z)-cover of L the lemma follows immediately.

We first prove that Q^* is an (S, C, D, Z)-cover of L. That Q^* consists of S plus elements, or cofaces of elements of Z, is immediate; thus it remains only to prove that Q^* is a cover of L, i.e., that $L \# Q^* = \overline{0}$ (see Lemma 3.5).

Now

$$\overline{0} = L \# Q^* \quad \text{by def } Q$$
$$= (L \# (Q - \{u'\})) \# u'$$

by the definition of the #-product and Lemma 3.7. But, by a similar argument and the definition of Q^*

$$L \# Q^* = (L \# (Q - \{u'\})) \# v'.$$

Thus, where $R = L \# (Q - \{u'\})$ it suffices to show that $R \# v' = \overline{0}$. Now if $R \# v' = T \neq \overline{0}$ then, since $R \# u' = \overline{0}$ it follows that T consists of faces of u' and that T is covered by $u' \# v'$ thus $u' \# v' \neq \overline{0}$. But then, since

$(u' \# (DUS) \# v' = \overline{0}$ it follows that $T \# (DUS) = \overline{0}$ hence, by the definition of DUS, T is part of L not covered by C and thus perforce T is covered by S. But S is in $Q-\{u'\}$ and so T cannot be covered by $R = L \# (Q-\{u\})$. This contradicts the definition of T. Thus we must have $R \# v' = \overline{0}$ and so $L \# Q^* = \overline{0}$ as desired. It remains to show that $c(Q^*) \leq c(Q)$. Recall that for any cover C, the cost, c(C) of C, is the cost of the reduction R(C) of C as given in section 3. Perforce u' is an output-coface of a cube p in R(Q) and v' is an output coface of a cube q in $R(Q^*)$.

Let U denote the set of all cofaces of elements of Q which are not cofaces of elements of $R(S \cup \{p\})$, then we see that

$$Q \approx S \cup \{u'\} \cup U$$

$$Q^* \approx S \cup \{v\} \cup U .$$

So, by proposition 2.1

$$c(Q) = c(S \cup \{u'\}) + c^*(S \cup \{u'\}, U)$$

$$c(Q) = c(S \cup \{v'\}) + c^*(S \cup \{v'\}, U) .$$

By assumption, $c(S \cup \{v'\}) \leq c(S \cup \{u'\})$ thus it suffices to show that

$$c^*(S \cup \{v'\}, U) \leq c^*(S \cup \{u'\}, U).$$

Let t denote the coface of q (if any) which appears in R(U). Let $V = R(U) - \{t\}$. Then, by proposition 2.1

$$c^*(S \cup \{v'\}, U) = c(S \cup \{v'\}, V) + c^*(S \cup \{v'\} \cup V, t)$$

and

$$c^*(S \cup \{u'\}, U) = c^*(S \cup \{u'\}, V) + c^*(S \cup \{u'\} \cup V, t)$$

$$= c^*(S \cup \{u'\}, V) + c^*(S \cup V, t)$$

by proposition 2.2 .

But $c^*(S \cup \{u'\}, V) = c^*(S \cup \{v'\}, V)$ by proposition 2.2, thus it suffices to show that

$$c^*(S \cup \{v'\} \cup V, t) \leq c^*(S \cup V, t) \qquad (1)$$

But

$$c^*(S \cup V, \{v'\} \cup \{t\}) = c^*(S \cup V, v') + c^*(S \cup V \cup \{v'\}, t)$$

by proposition 2.1, and by proposition 2.3 ,

$$c^*(S \cup V, \{v'\} \cup \{t\}) \leq c^*(S \cup V, v') + c^*(S \cup V, t)$$

which gives us (1) just as required.

Q. E. D.

Corollary 4.2: Given C, D, S and Z as described above and elements u and v in Z with $u < v$ then every minimum $(C, D, S, Z-\{u\})$-cover of L is also a minimum (C, D, S, Z)-cover of L.

Proof: The first part follows from the preceeding lemma. The second part is seen by noting that $w \# C = w$ implies that no vertex of w is covered by C and thus, conversely, w does not cover any vertex of C whence given any (C, D, S, Z)-cover Q of L with $w \in Q$ it follows that $Q - \{w\}$ is also a (C, D, S, Z)-cover of L and, since $c(Q-\{w\}) \leq c(Q)$, $Q-\{w\}$ costs no more than Q.

For use in the extraction algorithm we now define the less-than operation which, given S, C, D and Z applies the above result to eliminate elements from Z. More precisely, if

$$Z = a_1 \mid b_1,\ a_2 \mid b_2, \ldots, a_n \mid b_n,$$

then the result of the less than operation is $Z - Z^{\wedge}$ where $Z^{\wedge}$ is the subset of Z such that:

i) $a_i \mid b_i \in Z^{\wedge}$ if $a_i \mid b_i \# (D \cup S \cup (C - a_i \mid b_i)) \neq \overline{0}$

ii) $a_i \mid b_i \in Z^{\wedge}$ if there exists $j < i$ such that $a_i \mid b_i < a_j \mid b_j$

iii) $a_i \mid b_i \in Z^{\wedge}$ if there exists $j > i$ such that $a_i \mid b_i < a_j \mid b_j$ and it is not the case that $a_j \mid b_j < a_i \mid b_i$.

(The complications in rules ii and iii are necessary so that when both $u < v$ and $v < u$ we do not eliminate both u and v).

Definition: Given C, D, S and Z as above a cube e in Z is said to be an L-extremal, or simply an extremal with respect to (C, D, S, Z) if it covers a vertex of L not covered by any other cube in $Z \cup D \cup S$. Such a vertex is said to be distinguished.

Lemma 4.3: Let C, D, S and Z be as above, then a minimum (C, D, S, Z)-cover M of L contains every extremal e (with respect to (C, D, S, Z)) or one of its cofaces.

Proof: By definition an extremal covers a vertex of L not covered by any other cube of $Z \cup D \cup S$. Thus if M does not include $e = e_1 \mid e_2$ or one of its cofaces covering distinguished vertex w, then it must include a face $(f_1 \mid f_2) \sqsubset (e_1 \mid e_2)$. But the coface of $e = e_1 \mid f_2$, covers whatever $f_1 \mid f_2$ covers and has a lower cost (by assumption 1 on cost); thus $f_1 \mid f_2$ could not possibly belong to a minimum cover. Q. E. D.

In the single output case, an extremal itself must belong to every minimum cover. In the multiple-output case, however, as the above lemma shows, it can only be guaranteed that e or one of its output cofaces belongs to every minimum cover. The next step then in the extraction algorithm is the computation of the extremals of the problem.

Lemma 4.4: Given C, D, S and Z as above then $e \in Z$ is an extremal with respect to ((C, D, S, Z) if, and only if,

$$e \# (D \cup S \cup (Z-\{e\})) \neq \overline{0} .$$

Indeed, $e \# (D \cup S \cup (Z-\{e\}))$ will be a cover of the distinquished verticies corresponding to e.

Proof: Suppose e contains vertex $u|v$ not contained in any other prime cube of K and not a DON'T CARE condition. By successive applications of use of Lemma 3.2 this #-product is not empty. Conversely suppose $e \# (D \cup S \cup (z-e)) \neq \overline{0}$. Then by Lemmas 3.1 and 3.4 this #-product consists of a cover of all faces of e not in $D \cup S \cup (Z-e)$ so that e is by definition an extremal.

Q. E. D.

Lemma 4.5: Let P(e) denote the set of z in Z such that $z \sqcap e \neq \overline{0}$, excluding e itself, the periphery of e. Then e is an extremal if and only if

$$e \# (D \cup S \cup P(e)) \neq \overline{0} .$$

Note: This lemma should substantially reduce the amount of computation for the part of the algorithm computing the extremals.

Corollary 4.6: Given an extremal $e = e_1|e_2$ with respect to (C, D, S, Z) and given that

$$e \# (D \cup (Z-\{e\})) = \{a_1|b_1, \ldots, a_n|b_n\}$$

then

1) a minimum (C, D, S, Z)-cover M of L must contain the coface

$$\Delta(e, D, Z) = e_1|f = e_1|b_1 \sqcap b_2 \sqcap \ldots \sqcap b_n$$

of e, and

2) Every minimum

$$(-C, D, S \cup \{e_1|f\}, (Z-\{e_1|f\}) \cup \{e \# e_1|f\})$$

-cover of L is a minimum (C, D, S, Z)-cover of L.

Proof: 1. By the preceeding result $\{a_1|b_1, \ldots, a_n|b_n\}$ is a cover of the distinguished vertices corresponding to e. From 4.3 we know then that every minimum (C, D, S, Z)-cover must contain a coface of $e = e_1|e_2$ which covers $\{a_1|b_1, \ldots, a_n|b_n\}$. Clearly

$\Delta(e, D, Z) = e_1|f = e_1|b_1 \sqcap b_2 \sqcap \ldots \sqcap b_n$ is, by assumption 1 on cost, the cheapest possible such coface of e.

For use in the extraction algorithm we now define the #-extremal-operation to be the operation which, given S, C, D and Z, employs the method of Lemma 4.4 to find the set Z all the L-extremals with respect to (S, C, D, Z), by inspecting

each element of Z in turn. Where E is the set of L-extremals of L with respect to (S, C, D, Z) we define the Δ-operation to be the operation which, given E, S, C, D and Z, forms the set

$$\Delta(E, D, Z) = \{\Delta(e, D, Z) \mid e \in E\}$$

($\Delta(E,D, Z)$ as in Lemma 4.6), and we define the $\bar{\Delta}$-operation to be such that

$$\bar{\Delta}(E, D, Z) = \{e \# \Delta(e, D, Z) \mid e \in E\}$$

5. Description of F-notation and F-algorithms

We find it convenient to describe the extraction algorithm in a notation called the F-notation, F standing for "functional." This notation when made sufficiently precise actually serves to define the notion of algorithm. First we define an F-statement. F-statements are of two varieties, the execution statement and the conditional statement. The execution statement has the form

$$C = \underline{F}(A_1, \ldots, A_n)$$

where C is the range and the A_i are the domain-symbols and $\underline{F}$ is the function symbol, for either a primitive function or a previously defined function; the interpretation is that $\underline{F}$ is a function to be executed, A_i, its arguments and C, its value. Only a relative order of execution of the F-statements of an F-algorithm is specified by the F-algorithm: the actual sequencing of their execution might be determined by the computer executing the algorithm.

The conditional statement has the form

$$(f(A) = B \mid D = \underline{F}(E_1, \ldots, E_r))$$

this has the interpretation that if $f(A) = B$, f being another well-defined function, then the function $D = \underline{F}(E_1, \ldots, E_r)$ is to be executed.

An F-formula is composed of a string of F-statements in the following format: a left square bracket is followed by $\lceil(A_1,\ldots,A_r)$ where the A_i are the arguments of the F-formula, followed by the equality sign $\equiv$ followed by a string of F-statements each enclosed in round brackets, concluded with]. The informal interpretation is that $\lceil(A_1,\ldots,A_n)$ represents the function defined by its statement string. The F-statement of an F-formula may itself consist of the function $\Delta(B_1,\ldots,B_r)$ defined by another F-formula. In particular it might include the F-formula itself, for different arguments.

An F-algorithm is a string of F-formulas. The name of the F-algorithm $\underline{E}$ is written before the F-string in the form $E = \underline{E}(G_1,\ldots,G_n)$ where the G_i are the primitive arguments of the F-statements and F-formulae, i.e. arguments which are not themselves the values of F-formulas or F-statements.

The only requirement concerning the order of execution of F-formulas of an F-algorithm is a natural one: if F-formula A is used as a statement in F-formula B, then A must be completed before B is completed. Similarly, within an F-formula, statement S must be completed before statement T if statement T has an argument a value computed by statement S. In the absence of a rule specifying the order of execution, they are executed in order of occurrence.

We shall now describe the multiple-output extraction algorithm in this notation.

6. Description of the Multiple-Output Extraction Algorithm in F-Notation

In this section we present the multiple-output Extraction Algorithm using the F-notation as a means to tie together the various sub-algorithms and operations given in earlier sections of the paper. We first present the notations to be employed for representing the various sub-algorithms or primitives; we then present the extraction algorithm as an F-algorithm, and we follow this with a verbal description. Let

#alg	denote the #-algorithm for finding prime-cubes as given in section 3.
#ext	denote the #-extremal algorithm for finding extremals as given in section 3
Δ	denote the Δ-operation as in section 4
$\overline{\Delta}$	denote the $\overline{\Delta}$ operation as in section 4
<	denote the less-than operation as in section 4.

Finally let SEL denote an operation which, given Z, selects from Z an element f and an output coordinate i of f such that $f_i \neq X$.

In terms of these primitives the extraction algorithm may be written out as follows:

$$C_{min} = \underline{MX}(C, D, S)$$

1. $[\underline{MX}(C, D, S) = (C\#S = \overline{0} \mid S)(\ddot{Z} = \#alg(C \cup D))(\underline{E}(C, D, S, \ddot{Z}))\ ]$

2. $[\underline{E}(C, D, S, \ddot{Z}) = (C\#S = \overline{0} \mid S)(Z = <(D, S, \ddot{Z}))(E = \#ext(C, D, S, Z))$
$(E = \overline{0} \mid \underline{B}(C, D, S, Z))(F = \underline{\Delta}(E, D, S, Z))$
$(G = \overline{\Delta}(E, D, S, Z))(\underline{E}(C, D, S \cup F, (Z - E) \cup G))\ ]$

3. $[\underline{B}(C, D, S, Z) = ((a \mid b, i) = SEL(Z))(g = a \mid X \ldots \overset{i}{X} \ldots X)$
$(h = (a \mid b) \# g)(S^g = \underline{E}((C, D, S \cup \{g\}, (Z - \{a \mid b\}) \cup \{h\})$
$(S^g = \underline{E}(C, D, S, (Z - \{a \mid b\}) \cup \{h\}$
$(c(S^g) < c(\overline{S^g}) \mid S^g)(c(S^g) \geq c(\overline{S^g}) \mid S^g)\ ]$

The heading $C_{min} = \underline{MX}(C, D, S)$ states that the multiple output algorithm $\underline{MX}$ is a function of C, the CARE conditions, D the DON'T-CARE conditions and S the part of the minimum cover which is arbitrarily required to belong to the cover (classically, S is empty); the value of $\underline{MX}$ is the minimum cover C_{min} which it computes.

F-formula 1. consists of three F-statements: $(C \# S = \overline{0} \mid S)$ states that if $C \# S = \overline{0}$, that is, if S is already a cover then $\underline{MX}(C, D, S)$ has the value S, and the computation is completed. The second F-statement, $\ddot{Z} = \#alg(C \cup D)$, specifies that the #-algorithm (as given in section 3) is to be used for generating $\ddot{Z}$, the prime cubes of K from $C \cup D$. The third F-statement consists in an application of the F-formula $\underline{E}(C, D, S, Z)$, is described in F-formula 2. which will now be explained.

The first F-statement (in F-formula 2.) in $\underline{E}$ (the $\underline{E}$ standing for extraction) is again the test for completion: If $C \# S = \overline{0}$ then $C_{MIN} = S$. The second F-statement is the less-than operation (see section 4) applied to arguments D, S and $\ddot{Z}$. The resulting cover is denoted by Z. The algorithm now proceeds with Z replacing $\ddot{Z}$ in accordance to Lemma 4.2. The third statement says compute the extremals with respect to C, D, S, and Z and denote the result by E using the procedure established in Lemma 3.8 The fourth statement which is conditional says that if E is empty (i.e. $= \overline{0}$) then perform the operation $\underline{B}(C, D, S, Z)$ described by the F-formula 3. But if $E \neq \phi$ then the fifth and sixth F-statements direct one to perform the Δ and $\overline{\Delta}$ operations as described in section 4. The seventh and final F-statement in this F-formula says to repeat the application of $\underline{E}$ with new quantities indicated, thus replacing the original problem with a new one in accordance with Lemma 4.6 (thus recursion).

The third F-formula $\underline{B}$, the "Branch" formula, describes the procedure when $E = \overline{0}$, that is, when no extreme cubes are found. The procedure is to select an element, say $a \mid b$, of Z and one of its single-output cofaces $g = a \mid X \ldots X \overset{i}{1} X \ldots X$, to form two solutions, one S^g containing g, the other $S^{\overline{g}}$ not containing g and then to take $\underline{B}(C, D, S, Z)$ (and hence $\underline{E}(C, D, S, Z)$) to be the cheaper of S^g and $S^{\overline{g}}$ (where the costs are equal, $S^{\overline{g}}$ is arbitrarily chosen).

The proof that the algorithm always produces a minimum cover (under the assumptions on the cost function given in section 2) is given in the next section.

7. Proof of the Validity of the Multiple-Output Extraction Algorithm

The proof will refer to the "program" of the algorithm in F-notation and to the results proved in sections 2, 3 and 4.

Given a cover C of the CARE-complex, a cover D of the DON'T-CARE-complex, and a collection S of cubes required to be in the solution (in most practical problems $S = \phi$), the first step of the algorithm is to compute $Z = \#alg(C \cup D)$ in accordance with Theorem 3.3. From Lemma 2.4 it follows that every minimal (C, D, S, Z)-cover of K(C) is a minimal S-cover of K(C). Thus it remains only to show that for any (C, D, S, Z), $\underline{E}$ (C, D, S, Z) is a minimal S-cover of K(C).

Given a quadruple (C, D, S, Z) (satisfying the conditions given a the beginning of section 4) let the complexity of Z be be defined as the number of output coordinates having value 1, summing over all cubes of Z. The proof will now proceed by induction on the complexity, n, of Z.

Say $n = 1$, so $Z = \{z\}$. Referring to the algorithm we see that if $C\#S = \overline{0}$ then $\underline{E}(C, D, S, Z) = S$ while if $C\#S \neq \overline{0}$ then z must be an L-extremal (L=K(C))with respect to (C, D, S, Z) for, by assumption $S \cup Z = S \cup \{z\}$ is a cover of L and $z\#(D \cup S \cup (Z - \{z\})) = z\#(D \cup S)$ whence $z\#(D \cup S) = \overline{0}$ would imply that there is no part of L not covered by z. Thus we have $E = \{z\}$. But since the complexity of z is 1 is clear that $\underline{\Delta}$ (E, D, S, Z) = {z}, and $\overline{\underline{\Delta}}$ (E, D, S, Z) = ϕ. Hence the algorithm states that

$$\underline{E}\ (C, D, S, Z) = \underline{E}(C, D, S \cup \{z\},\ \phi).$$

But, since $C\#(S \cup \{z\}) = \overline{0}$ it follows then, from the first F-statement in $\underline{E}$ that

$$E\,(C, D, S, Z) = \{z\};$$

which is clearly the minimal S-cover.

Now assume the result is proved for complexity up to $n = k \geq 1$ and consider the case where Z has complexity $k+1$.

If $C\#S = \overline{0}$ then $\underline{E}(C, D, S, Z) = S$ which is clearly then a minimum S-cover for L. If $C\#S \neq \overline{0}$ suppose that the less-than operation deletes cubes from Z and thus replaces Z with, say $Z^{\wedge} \subsetneq Z$. Clearly the algorithm then proceeds just as it would have if we had started with $Z = Z^{\wedge}$, i.e. we have

$$\underline{E}\ (C, D, S, Z) = \underline{E}\ (C, D, S, Z^{\wedge}).$$ Now $Z^{\wedge} \subsetneq Z$ implies that the complexity of $Z^{\wedge}$ is less than that of Z. Thus form the induction hypothesis it follows that $\underline{E}(C, D, S, Z^{\wedge})$ is a minimal $(C, D, S, Z^{\wedge})$-cover of L and from theorem 4.2 it follows that every minimum $(C, D, S, Z^{\wedge})$-cover of L is a

minimum (C, D, S, Z)-cover of L. Thus $\underline{E}(C, D, S, Z)$ is a minimum (C, D, S, Z)-cover, and hence a minimum S-cover, of L. Thus the algorithm works if the less-than operation deletes cubes from Z.

Let us now assume that the less than operation does not delete any cubes from Z. In this case the algorithm proceeds to compute the set E of L-extremals with respect to (C, D, S, Z). Assuming $E \neq \overline{0}$ the algorithm then states that

$$\underline{E}(C, D, S, Z) = \underline{E}(C, D, S \cup F, (Z-E) \cup G).$$

Now if $E \neq \overline{0}$ then clearly the complexity of $(Z-E) \cup G$ is less than that of Z so, by the induction hypothesis, $\underline{E}(C, D, S \cup F, (Z-E) \cup G)$ is a minimal $(C, D, S \cup F, (Z-E) \cup G)$-cover of L is a minimum (C, D, S, Z)-cover, and hence a minimum S-cover of L. Thus the validity of the algorithm is established by the case where $E \neq \overline{0}$.

Finally let us assume $E = \overline{0}$. Then the algorithm says

$$\underline{E}(C, D, S, Z) = \underline{B}(C, D, S, Z)$$

Now in $\underline{B}$ the algorithm computes picks a coface $g = a|X \ldots \overset{i}{X}1X \ldots X$ of an element $a|b$ of Z and computes

$S^g = \underline{E}(C, D, S \cup \{g\}, (Z-\{a|b\}) \cup h)$ $(h = a|b \# g)$ and

$S^{\overline{g}} = \underline{E}(C, D, S, (Z-\{a|b\}) \cup \{h\})$.

Then the algorithm compares the cost of S^g and $S^{\overline{g}}$ and

$$E(C, D, S, Z) = \begin{cases} S^g & \text{if } c(S^g) < c(S^{\overline{g}}) \\ S^{\overline{g}} & \text{if } c(S^g) \geq c(S^{\overline{g}}) \end{cases}$$

Now the complexity of $(Z-\{a|b\}) \cup \{h\}$ is one less than that of Z hence, by the induction hypothesis, S^g is a minimum $(C, D, S \cup \{g\}, (Z-\{a|b\}) \cup \{h\})$-cover of L and $S^{\overline{g}}$ is a minimum $(C, D, S, (Z-\{a|b\}) \cup \{h\})$-cover of L. But clearly, S^g is a minimum S-cover of L <u>containing</u> g while $S^{\overline{g}}$ is a minimum S-cover of L <u>not containing</u> g. Clearly then the cheaper of S^g and $S^{\overline{g}}$ is a minimum S-cover of L and thus, in all cases, the algorithm produces a cover of minimum cost for complexity k+1. Hence, by induction, the algorithm always produces an S-cover of minimum cost.

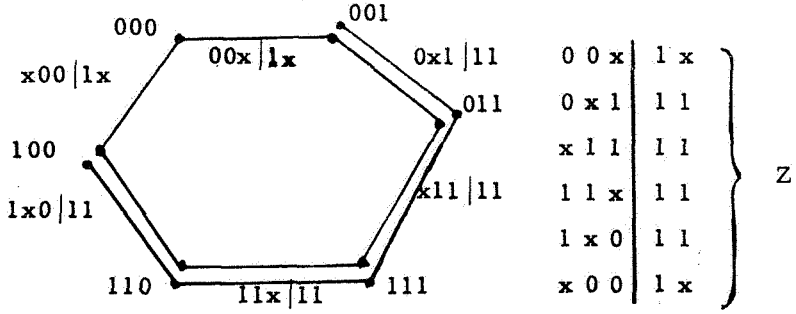
8. Concerning the Programming of the M. O. Extraction Algorithm

The programming aspects of the algorithm will be discussed with reference to the F-notation description. The F-notational description and the various auxiliary descriptions of sub-algorithms make the formal understanding of the algorithm more ready of access for programming than the single output algorithm. The F-notational description deliberately suppressed any of the details of storing of partially formed solutions S nor of keeping "track" of the branching that would be necessary for the solution of some problems.

The algorithm #alg for computing the prime cubes as given in Section 4 would be a fairly direct undertaking. A program for the entire algorithm has been written on the APL\360 system, to be reported on elsewhere [RWL].

Appendix I: Example of Execution of M. O. Extraction Algorithm

The example is a 3-input, 2-output problem depicted in the figure below. Two "overlapping" cubical complexes are used to depict the problem. To the right is a listing Z of all the prime cubes for the problem. The highlights of the



0 0 x	1 x
0 x 1	1 1
x 1 1	1 1
1 1 x	1 1
1 x 0	1 1
x 0 0	1 x

Z

computation will be given. This problem as it stands has two extremals as shown by the following computations.

	0 x 1	1 1
#	0 0 x	1 x
	0 x 1	x 1
	0 1 1	1 1
#	x 1 1	1 1
	0 0 1	x 1

$\Rightarrow$ 0 x 1 | 1 1 is an extremal, with

0 x 1 | x 1 as its distinguished coface

Similarly that $1\,x\,0\,|\,1\,1$ is an extremal, with $1\,x\,0\,|\,x\,1$ as its distinguished coface, is shown by the following computation

	$1\,x\,0$	$1\,1$
#	$x\,0\,0$	$1\,x$
	$1\,x\,0$	$x\,1$
	$1\,1\,0$	$1\,1$
#	$1\,1\,x$	$1\,1$
	$1\,0\,0$	$x\,1$

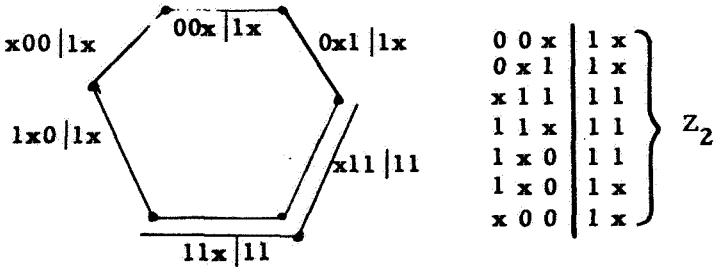
$\Rightarrow$ $1\,x\,0\,|\,1\,1$ and $1\,x\,0\,|\,x\,1$ is extreme and belongs to a minimum cover

On the other hand, that $0\,0\,x\,|\,1\,x$ is not an extremal is shown by the computation

	$0\,0\,x$	$1\,x$
#	$0\,x\,1$	$1\,1$
	$0\,0\,0$	$1\,x$
#	$x\,0\,0$	$1\,x$
	$\overline{0}$	$1\,x$

$\equiv \overline{0}$

Thus F_1 consists of $0\,x\,1\,|\,x\,1$ and $1\,x\,0\,|\,x\,1$. Thus the new set Z_2 of prime cubes consists of the list on the right



$0\,0\,x$	$1\,x$
$0\,x\,1$	$1\,x$
$x\,1\,1$	$1\,1$
$1\,1\,x$	$1\,1$
$1\,x\,0$	$1\,1$
$1\,x\,0$	$1\,x$
$x\,0\,0$	$1\,x$

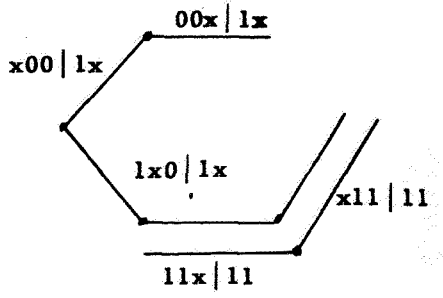
$\} Z_2$

depicted by the diagram on the left above.

The next operation to be performed is the $<$- operation. This is performed on the cubes of Z_2. We recall that $a < b$ if cost $a \geq$ cost b and if $a \sqsubset b \cup D$. It is seen by inspection that there are no cubes of Z_2 "dominated" by others. Consequently we must execute the branch operation $\underline{B}$.

Let us suppose that the selection operation SEL chooses $f = 0\,x\,1\,|\,1\,x$ for branching; here the distinguished coordinate $i = 1$ and $g = f$ while $h = \overline{0}$. Accordingly we must compute two solutions S^g containing g and $S^{\overline{g}}$ not containing g.

Computation of S^g. The partial solution S_1 thus far developed consists of $0\,x\,1\,|\,x\,1$ and $1\,x\,0\,|\,x\,1$. Including $0\,x\,1\,|\,1\,x$ yields the partial solution $0\,x\,1\,|\,1\,1$ and $1\,x\,0\,|\,x\,1$. The remaining complex is depicted below.



The don't care complex D_2 then consists of S_1. First we show that $0\ 0\ x\,|\,1\ x < x\ 0\ 0\,|\,1\ x$.

For $(0\ 0\ x\,|\,1\ x)\ \#\ (0\ x\ 1\,|\,1\ 1) = 0\ 0\ 0\,|\,1\ x \sqsubset x\ 0\ 0\,|\,1\ x$.

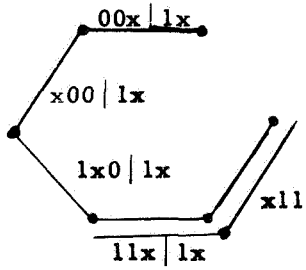
$x\ 0\ 0\,|\,1\ x$ becomes an extremal. Similarly

$x\ 1\ 1\,|\,1\ 1 < 1\ 1\ x\,|\,1\ 1$, which results in making $1\ 1\ x\,|\,1\ 1$ a (2nd order) extremal. This completes the formation of S^g

$$\left.\begin{array}{l|l} 0\ x\ 1 & 1\ 1 \\ 1\ x\ 0 & x\ 1 \\ x\ 0\ 0 & 1\ x \\ 1\ 1\ x & 1\ 1 \end{array}\right\} S^g$$

whose cost is 8 input lines and 6 output lines.

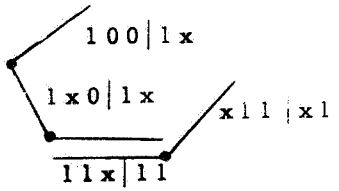
Formation of $S^{\bar{g}}$. In this operation $g = 0\ x\ 1\ |\ 1\ x$ is deleted from Z. In this event since $(0\ 0\ x\,|\,1\ x)\ \#\ (x\ 0\ 0\,|\,1\ x) = (0\ 0\ 1\,|\,1\ x)$, $0\ 0\ x\,|\,1\ x$ becomes an extremal and the computation



$$\begin{array}{rl|l} & x\ 1\ 1 & 1\ 1 \\ \# & 1\ 1\ x & 1\ 1 \\ \hline & 0\ 1\ 1 & 1\ 1 \\ \# & 0\ x\ 1 & x\ 1 \\ \hline & 0\ 1\ 1 & 1\ x \end{array} \in D_2$$

shows that $x\ 1\ 1\ |\ 1\ 1$ is an extremal with $x\ 1\ 1\ |\ 1\ x$ being adjoined to the solution. The solution thus far consists then of

$$\begin{array}{l|l} 0\ x\ 1 & x\ 1 \\ 0\ 0\ x & 1\ x \\ x\ 1\ 1 & 1\ x \\ 1\ x\ 0 & x\ 1 \end{array}$$



Here in the remaining complex

$$x\ 0\ 0\ |\ 1\ x < 1\ x\ 0\ |\ 1\ x$$

$$x\ 1\ 1\ |\ x\ 1 < 1\ 1\ x\ |\ 1\ 1$$

Now

$$\begin{array}{rl|l} & 1\ x\ 1 & 1\ x \\ \# & 1\ 1\ x & 1\ 1 \\ \hline & 1\ 0\ 0 & 1\ x \\ \# & 1\ x\ 0 & x\ 1 \\ \hline & 1\ 0\ 0 & 1\ x \end{array} \Rightarrow 1\ x\ 1\ |\ 1\ x \text{ is extremal}$$

Similarly

$$\begin{array}{rl|l} & 1\ 1\ x & 1\ 1 \\ \# & 1\ x\ 0 & 1\ 1 \\ \hline & 1\ 1\ 1 & 1\ 1 \\ \# & x\ 1\ 1 & 1\ x \\ \hline & 1\ 1\ 1 & x\ 1 \end{array}$$

$1\ 1\ x\ |\ 1\ 1 \Rightarrow$ is extremal with $1\ 1\ x\ |\ x\ 1$ adjoined to solution

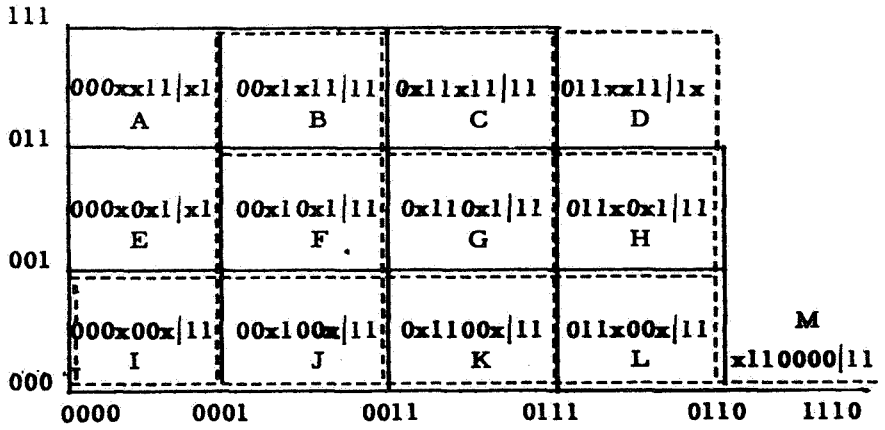
Solution $S^{\bar{g}}$

$$\left.\begin{array}{l|l} 0\ x\ 1 & x\ 1 \\ 1\ x\ 0 & x\ 1 \\ 0\ 0\ x & 1\ x \\ x\ 1\ 1 & 1\ x \\ 1\ x\ 0 & 1\ x \\ 1\ 1\ x & x\ 1 \\ \hline 0\ x\ 1 & x\ 1 \\ 1\ x\ 0 & 1\ 1 \\ 0\ 0\ x & 1\ x \\ x\ 1\ 1 & 1\ x \\ 1\ 1\ x & x\ 1 \end{array}\right\} S^{\bar{g}}$$

Here $S^{\bar{g}}$ has cost of 10 input lines plus 6 output lines thus S^g is the minimum solution.

Appendix II: Another Example of the Extraction Algorithm

The geometric picture below depicts a 7-input 2-output problem. The complex associated with the first output is given as the solid-line figure while that for the second, by the dotted line figure. It may be observed that the figure faithfully represents the complex in the sense that only the prime



cubes are shown but if two prime cubes have an interface then the figure correctly represents this face. Some details of the computations will be given.

Initially we start with no partial solution, $S_o = \overline{0}$. We shall assume that the prime cubes, A through M, have been priorly computed. Furthermore we shall frequently take advantage of the geometric figure to shorten the computations e.g. for extremals, a la Lemma 4.5.

That A is an extremal follows from the following computations:

A	0 0 0 x x 1 1	x 1
B#	0 0 x 1 x 1 1	1 1
	0 0 0 0 x 1 1	x 1
E#	0 0 0 x 0 x 1	x 1
	0 0 0 0 1 1 1	x 1

Similarly that D is an extremal follows from the following computation:

D	0 1 1 x x 1 1	1 x
C#	0 x 1 1 x 1 1	1 x
	0 1 1 0 x 1 1	1 x
H#	0 1 1 x 0 x 1	1 1
	0 1 1 0 1 1 1	1 x
G#	0 x 1 1 0 x 1	1 1
	0 1 1 0 1 1 1	1 x

In similar fashion we compute the following list of first-order extremals or, more precisely, their distinguished cofaces.

$F_1 = S_1$

	A	0 0 0 x x 1 1	x 1
δ_2	B	0 0 x 1 x 1 1	1 x
δ_1	C	0 x 1 1 x 1 1	x 1
	D	0 1 1 x x 1 1	1 x
δ_1	H	0 1 1 x 0 x 1	x 1
	I	0 0 0 x 0 0 x	1 1
	M	x 1 1 0 0 0 0	1 1

This leaves Z_2 as

$B_2 = B \# \delta_2 B$	0 0 x 1 x 1 1	x 1
C_2	0 x 1 1 x 1 1	1 x
E^2	0 0 0 x 0 x 1	x 1
F	0 0 x 1 0 x 1	1 1
G	0 x 1 1 0 x 1	1 1
$\delta_2 H$	0 1 1 x 0 x 1	1 x
J^2	0 0 x 1 0 0 x	1 1
K	0 x 1 1 0 0 x	1 1
L	0 1 1 x 0 0 x	1 1

The following computation shows that $J < K$

J	0 0 x 1 0 0 x	1 1
I#	0 0 0 x 0 0 x	1 1
	0 0 1 1 0 0 x	1 1 $\sqsubset$ K
K	0 x 1 1 0 0 x	1 1

and the computation

K	0 x 1 1 0 0 x	1 1
G#	0 x 1 1 0 x 1	1 1
	0 x 1 1 x x x	1 1
L#	0 1 1 x 0 0 x	1 1
	0 0 1 1 0 0 0	1 1

shows that K is a (2^{nd} order) extremal. Similarly, $\delta_1 H > L$ is shown to be a (3^{rd} order) extremal and we are left with a final solution

F_3

	A	0 0 0 x x 1 1	x 1
δ_2	B	0 0 x 1 x 1 1	1 x
δ_1^2	C	0 x 1 1 x 1 1	x 1
	D	0 1 1 x x 1 1	1 x
	H	0 1 1 x 0 x 1	1 1
	I	0 0 0 x 0 0 x	1 1
	K	0 x 1 1 0 0 x	1 1
	M	x 1 1 0 0 0 0	1 1

Appendix 3 Example indicating comparison of number of prime cubes (implicants) of singular notation and Muller encoding.

Definition of problem using singular cubes

inputs a b c d	outputs 1 2 3 4	
1 1 x 1	1 x x 1	} care conditions
x 1 1 1	x 1 1 x	
0 0 0 0	1 1 1 1	

Definition of problem using Muller encoding

	a b c d	1 2 3 4	
1	x x x x	0 0 x x	} don't care conditions introduced by encoding
2	x x x x	0 x 0 x	
3	x x x x	0 x x 0	
4	x x x x	x 0 0 x	
5	x x x x	x 0 x 0	
6	x x x x	x x 0 0	
7	x x x x	1 1 1 1	
1	1 1 x 1	1 x x x	} care conditions
2	1 1 x 1	x x x 1	
3	x 1 1 1	x 1 x x	
4	x 1 1 1	x x 1 x	
5	0 0 0 0	1 x x x	
6	0 0 0 0	x 1 x x	
7	0 0 0 0	x x 1 x	
8	0 0 0 0	x x x 1	

Partial computation of prime cubes (prime implicants)

*-product	1 1 x 1 1 x x x
x x x x 0 0 x x	1 1 x 1 x 0 x x
x x x x 0 x 0 x	1 1 x 1 x x 0 x
x x x x 0 x x 0	1 1 x 1 1 x x 0
x x x x x 0 0 x	subsumed
x x x x x x 0 0	subsumed

This computation shows that the *-product of the first care cube listed above introduces eight new prime cubes, and similarly for each of the eight. Thus $8 \times 3 = 24$ new prime cubes are so formed. Now each of the 7 pure don't-care and 8 pure care cubes originally listed are also prime. Thus the total number of prime cubes for the problem described in the Muller encoding is

$$7 + 8 + 24 = \underline{\underline{39}} ,$$

as compared with $\underline{\underline{3}}$ in the singular notation.

REFERENCES

[B] Bartee, T. C., "Computer Design of Multiple Output Logical Networks," IRE Trans. on Elec. Computers, Vol. EC-10 (1961), pp. 21-30.

[Q] Quine, V. W., "The Problem of Simplifying Truth Functions," Amer. Math. Monthly, Vol. 59 (1952) pp. 521-531.

[ERW] Ann C. Ewing, E. G. Wagner, J. Paul Roth, "Algorithms for Logical Design", AIEE Transactions on Communications and Electronics, Vol. 56, pp. 450-458.

[Mi] Raymond E. Miller, "Switching Theory, Vol. 1, "Combinational Circuits", John Wiley & Sons, Inc., New York, 1965.

[Mu] David E. Muller, "Application of Boolean Algebra to Switching Circuit Design and to Error Detection," IRE Transactions on Electronic Computers, pp. 6-12, September 1954.

[R-1] J. Paul Roth, "Algebraic Topological Methods in Synthesis," Proceedings of an International Symposium on the Theory of Switching, April 2-5, 1957, in Annals of Computation Laboratory of Harvard Univ. Vol. 29, pp. 543-558, November 1960.

[R-2] J. Paul Roth, "Minimization over Boolean Trees," IBM Journal of Research and Development, Vol. 14, N. S. pp. 543-558, November 1960.

[RP] J. Paul Roth and M. Perlman, Space Applications of a Minimization Algorithm, IEEE Transactions on Aerospace and Electronic Systems. To appear.

[RWL] J. Paul Roth, Eric G. Wagner, Leon S. Levy, "An Algorithm and a Program for the Multiple-Output 2-Level Logic-Minimization Problem. To appear.

AN AXIOMATIC TREATMENT OF ROTH'S EXTRACTION ALGORITHM*

by

Eric G. Wagner

IBM Watson Research Center
Yorktown Heights, New York

ABSTRACT: In this paper, we present a general axiomatic treatment of J. Paul Roth's "extraction algorithm" for the minimization of logical circuits. This treatment brings together the seemingly different versions of the algorithm presented in Roth's different papers. It provides a general proof of the algorithm over a wide range of cost functions. The minimization problem and the algorithm are presented in an abstract context (i. e., without direct reference to any particular application, such as switching circuits) and is thus applicable to any "covering problem" in which the axioms are satisfied.

* This research was supported in part by J. P. L. Contract #951-538.

INTRODUCTION

In this paper we present a general axiomatic treatment of J. P. Roth's "extraction algorithm." This treatment brings together the seemingly different versions of the algorithm presented in Roth's different papers, it provides a general proof of the algorithm, and it facilitates the application of the algorithm to new situations.

The extraction algorithm was originally developed by Roth [R-2] as a means (algorithm) for finding minimal two-level AND-OR circuits. In later papers it was refined [E-R-W] and special extraction algorithms were developed for other classes of logical circuits [R-W] (and various unpublished results).

The type of problem to which the extraction algorithms are directed may be roughly described as follows: We are given a finite set K of objects which (in some sense) cover another set of objects L and, indeed, cover it more than once. Each object in K has a non-negative cost associated with it. The extraction problem is to find a subset M of K which covers L and is of minimal cost in that there is no other subset of K which both covers

L and is of lower cost than M. In the simpler cases the cost of a subset of K is just the sum of the cost of its elements; however, in the general case, more complex cost functions are employed.

From a pure mathematics point of view, the extraction problem is trivial since, because K is finite, the problem may always be solved by an exhaustive examination and comparison of all subsets of K. Clearly though, if K is large, the number of subsets is astronomical (e.g., if K contains 20 elements, then there are approximately 1,000,000 subsets), and such an exhaustive examination is impractical using even the fastest computers. Thus, the real problem is to develop algorithms which are efficient enough to deal with problems of reasonable size. The efficacy of the general extraction algorithm is, as will be seen, dependent on the nature of the problem. In the worst case it approaches exhaustion, but in the best cases it provides an answer directly without any exhaustive examination of cases. Design automation programs employing the extraction algorithm have proven their usefulness in the field in application to a variety of

real design problems.

In this paper our treatment of the extraction problem and the extraction algorithm is quite abstract (and thus quite general). We start, in Section 1, by stating the extraction problem in terms of abstractly defined notions of cover and cost. (The notion of a cover is defined in terms of a "difference" operation.) In Section 2 we present a first set of axioms for the extraction algorithm. These axioms are quite complex (to the point of inelegance), but they lead to a very general presentation and proof of the extraction algorithm. In Section 3 we present an alternative set of axioms and show that they imply the first set. These simpler axioms are designed to facilitate the proof (or disproof) of the applicability of the extraction algorithm in real situations. In Section 4 we sketch two examples of applications of the algorithm.

Notational Conventions

R^+ non-negative real numbers;

$K - k$ for $K - \{k\}$;

$K \cup k$ for $K \cup \{k\}$;

$P(S)$ power set of S (set of all subsets of S);

Π a partition of K, $k \in K$, $\Pi(k)$ class in Π containing k (partitions do not include ϕ).

1. THE EXTRACTION PROBLEM

In this section we present the basic definitions used in our approach. We will give interpretations of these definitions, but their "real meaning" is given by the axioms in the remaining sections. We start from:

T , a set (from which we draw subsets to be covered);

S , a set (from which we draw the subsets which cover);

$d : P(T) \times P(T \cup S) \rightarrow P(T)$, the <u>difference function</u>;

$c : P(S) \times P(S) \rightarrow R^{+}$, the <u>relative-cost function</u>.

Informally speaking, what we are interested in is "covering" subsets of S with subsets of T of minimal "cost." The notion of "covering" is defined in terms of the function d ; the notion of "cost" is defined in terms of the function c . Given $L \subset T$ and $K \subset S$, we can interpret $d(L, K)$ as being "the part of L <u>not</u> covered by K ." Correspondingly, we can interpret $d(L, d(L, K))$

as being "the part of L covered by K." Given K, $K' \subset S$ we can interpret $c(K, K')$ as being "the cost of K' given that one already has K." Of course, these interpretations will not "make sense" for arbitrary choices of d and c. However, with the axioms given in the following sections, these interpretations become "natural." These interpretations though lead to the following definitions:

Let $c^* : P(S) \rightarrow R^+$ such that, for every $K \subset S$,

$$c^*(K) = c(\phi, K) .$$

Given $K \subset S$, $L \subset T$, and $I \subset S$, we define a

<u>cover</u> of L to be any subset $C \subset S$ such that

$$d(L, C) = \phi ;$$

<u>(K/I)-cover</u> of L to be any cover C of L such that $I \subset C \subset K \cup I$;

<u>K-cover</u> of L to be any (K/ϕ)-cover of L;

<u>minimal (K/I)-cover</u> of L to be a (K/I)-cover M of L such that, for every (K/I)-cover

C of L

$$c^*(C) \geq c^*(M) ;$$

minimal K-cover of L to be any minimal (K/ϕ)-cover of L .

Using the above definitions, we define:

The Extraction Problem: Given T, S, d, and c , and given $L \subset T$ and $K \subset T$, K finite, and such that $d(L, K) = \phi$ (i. e., K is a cover of L) , find a minimal K-cover of L .

2. THE BASIC EXTRACTION ALGORITHM--FIRST AXIOMS AND PROOF

This section begins with four rather complex axioms which we may impose on T, S, d and c. We then present an algorithm, the extraction algorithm, and show that, when the axioms hold, this algorithm always leads to a solution of the extraction problem. The complexity of the axioms facilitates the statement and proof of the extraction algorithm; in the next section we will present a variety of simpler axioms which imply these initial axioms.

The Initial Axioms

While the extraction problem was stated purely in terms of T, S, d, and c, the axioms and algorithm employ one additional object, namely, a partition Π of S. Given any $K \subset S$, let Π_K denote the restriction of Π to K, and, given $k \in K \subset S$, let $\Pi_K(k)$ denote the element of Π_K which contains k. The axioms on T, S, d, c, and Π are then as follows:

For all $L \subset T$ and $I, K \subset S$ such that $d(L, I \cup K) = \phi$:

Axiom 1: If $k \in K$ and $d(L, I \cup (K-k)) \neq \phi$, then k is in every minimal (K/I)-cover of L.

Axiom 2: If M is a minimal (K/I)-cover of L and $k \in M - I$, and Q is a minimal $((K-k)/(I \cup k))$-cover of $d(L, k)$, then Q is a minimal (K/I)-cover of L.

Axiom 3: If k, $k' \in J = K - I$, with $\Pi_J(k) \neq \Pi_J(k')$ and if $d(d(L, d(L, I \cup \Pi_J(k)), I \cup \Pi_J(k')) = \phi$ and if $c(I, \Pi_J(k')) \not\leq c(I, \Pi_J(k))$, then every minimal $((K-\Pi_J(k))/I)$-cover of L is also a minimal (K/I)-cover of L, and there exists at least one minimal $((K-\Pi_J(k))/I)$-cover of L.

Axiom 4: For all $L \subset T$, $d(L, \phi) = \phi$.

The Extraction Algorithm

Given I, $K \subset S$ and $L \subset T$, the following algorithm defines an object $M(L, K/I)$; the theorems following the algorithm show that this is the desired minimum cover under appropriate conditions. We assume,

for expositional convenience, that a linear ordering is given on K.

START: go to 1.

1. Let $J = K - I$, does there exist a pair $\langle k, k' \rangle \in J \times J$ with $\Pi_J(k) \neq \Pi_J(k')$, but with

$$d(d(L, d(L, I \cup \Pi_J(k))), I \cup \Pi_J(k')) = \phi$$

and with $c(I, \Pi_J(k')) \leq c(I, \Pi_J(k))$? If yes, go to 2; if no, go to 3.

2. Let $\langle k, k' \rangle$ be the least such pair (under the lexicographical ordering of $K \times K$ induced by the linear ordering on K), then take

$$M(L, K/I) = M(L, (K - \Pi_J(k))/I) .$$

3. Does there exist any element $k \in K - I$ such that

$$d(L, I \cup (K-k)) \neq \phi \ ?$$

If yes, go to 4; if no, go to 5.

4. Let k be the first such element (under the linear ordering on K). If $d(L, I \cup k) = \phi$, then take

$$M(L, K/I) = I \cup k$$

and stop; otherwise, take

$$M(L, K/I) = M(d(L, k), (K-k)/(I \cup k)) .$$

5. Pick $k \in K$ (say the first element) and compute

$$A = M(d(L, k), (K-k)/(I \cup k))$$

and $B = M(L, (K-k)/I)$. If $c^*(A) > c^*(B)$, then take $M(L, K/I) = B$; if $c^*(B) \geq c^*(A)$, then take $M(L, K/I) = A$.

Theorem 2.1: If $I, K \subset S$ and $L \subset T$ such that $d(L, I \cup K) = \phi$, but $d(L, I) \neq \phi$, K is finite, and Axioms A.1, A.2, A.3, and A.4 hold, then the result $M(L, K/I)$ of the extraction algorithm is a minimal (K/I)-cover of L .

Proof: We proceed by induction on the size (number of elements in) K .

Say that K contains $n = 1$ elements so $K = \{k\}$. Since, by assumption $d(L, I) \neq \phi$, it is clear that $I \cup \{k\}$ is the minimal (K/I)-cover of L . Now consider the application of the algorithm. Since $K = \{k\}$, it is clear

that $\Pi = \{\{k\}\}$ and that step 1 carries us to step 3. But since $K - k = \phi$, we then have

$$d(d(L, d(L, I \cup k), I \cup (K-k)) = d(d(L, \phi), I \cup \phi) =$$

$$d(L, I) \text{ by A. 4}$$

$\neq \phi$ by theorem statement.

Thus we go to step 4 where, since $d(L, I \cup k) = \phi$, we stop with $M(L, K/I) = I \cup \{k\}$, which is just what we desired.

Assume now that the result has been proved for all I, K, and L where K has n ($n \geq 1$) or fewer elements. Consider I, K, and L where K has n + 1 elements. We consider three cases:

Case 1: There exist $k, k' \in J = K - I$ satisfying A. 3. That is, $\Pi_J(k) \neq \Pi_J(k')$, but

$$d(d(L, d(L, I \cup \Pi_J(k))), I \cup \Pi_J(k')) = \phi$$

and $c(I, \Pi_J(k')) \leq c(I, \Pi_J(k))$. Then, by A. 3, there exists a minimal $((K-\Pi_J(k))/I)$-cover M of L which is a mini-

mal (K/I)-cover of L. But, turning to the algorithm we see then that step 1 will carry us to step 2 (since the desired k, $k' \in J$ exist). Now step 2 makes

$$M(L, K/I) = M(L, (K-\Pi_J(k))/I) .$$

But, since $K - \Pi_J(k)$ is smaller than K, it follows from the induction hypothesis, that $M(L, (K-\Pi_J(k))/I)$ is a minimal $((K-\Pi_J(k))/I)$-cover of L, and thus, by the above $M(L, K/I)$ is a minimal (K/I)-cover of L.

Case 2: There do not exist k, $k' \in K - I$ satisfying A. 3 but there exists $k \in K$ satisfying A. 1; that is,

$$d(d(L, d(L, I \cup k)), I \cup (K-k) \neq \phi .$$

Then in this case we know, by A. 1, that k is in every minimal (K/I)-cover M of L. But then Axiom A. 2 applies, that is, if Q is any minimal $((K-k)/I \cup k)$-cover of $d(L)$, then Q is a minimal (K/I)-cover of L. But, turning to the algorithm, we see

that step 1 carries us to step 3 which will carry us to step 4 (since $k \in K$ exists satisfying A. 2). Now step 4 makes

$$M(L, K/I) = M(d(L, k), (K-k)/(I \cup k)) .$$

But since $K - k$ is smaller than K, it follows from the induction hypothesis that $M(d(L, k), (K-k)/(I \cup k))$ is a minimal $((K-k)/(I \cup k))$-cover of $d(L, k)$, and thus, by the above, $M(L, K/I)$ is a minimal (K/I)-cover of L.

Case 3: There do not exist k, $k' \in K - I$ satisfying A. 3 or A. 1. Then clearly, if we pick $k \in K$, then either there exists a minimal (K/I)-cover M of L including k or there does not. If a minimal (K/I)-cover M exists with $k \in M$, then, by A. 2, every $((K-k)/I \cup k)$-cover of $d(L, k)$ is a minimal (K/I)-cover of L. On the other hand, if no such minimal (K/I)-cover exists, then there must exist a minimal (K/I)-cover M with $k \notin M$. (Note that since A. 1 does not hold, there exist

(K-k/I)-covers of L.) But then this cover M is clearly a ((K-k)/I)-cover of L. Now, turning to the algorithm, we see that step 1 carries us to step 3 which carries us to step 5. But then we take M(L, K) to be the cheaper of

$$A = M(d(L, k), ((K-k)/(I \cup k))$$

and

$$B = M(L, (K-k)/I) .$$

But since K - k is smaller than K, it follows from the induction hypothesis that these are the desired minimal covers.

Since Case 3 is essentially an exhaustive algorithm, it is clear that these three cases cover all possibilities and thus it follows, by induction, that the algorithm always produces a minimal cover. Q. E. D.

Corollary 2.2: If $K \subset S$ and $L \subset T$ such that $d(L, K) = \phi$, $L \neq \phi$, K is finite, and Axioms A.1, A.2, A.3 and A.4 hold, then the result $M(L, K/\phi)$ of the extraction algorithm is a minimal K-cover of L.

Proof: Follows immediately from the preceding theorem by taking $I = \phi$. Q. E. D.

3. ALTERNATIVE AXIOMS

In this section we will present some alternative axioms for the extraction algorithm. These axioms will imply the axioms given in the preceding section, but they are not strictly equivalent to them (i. e., they are not implied by the earlier axioms). In the first part of the section, we present axioms for the "difference function" d. These axioms are sufficient to prove Axioms A. 1, A. 2, and A. 4 of the preceding section (indeed, they include Axiom A. 4). In the second part of the section, we present axioms on the cost function and employ them to prove Axiom A. 3.

Axioms for the Difference Function

We start by defining a relation $\equiv$ on $P(T)$. Given L, $L' \subset T$ we write $L \equiv L'$ if and only if

$$d(L, L') = d(L', L) = \phi .$$

Intuitively, $L \equiv L'$ means are two representations of the same thing--i. e., it will be the case that anything

which covers L also covers L' and vice versa. Note that $\equiv$ is, by definition, a symmetric relation, but until we impose further properties on d, it is not necessarily either reflexive or transitive and thus the above intuitive interpretation is dependent on the axioms given for d.

The Difference Axioms (or D-axioms) are as follows:

Axiom D.1: For all $L \subset T$, $d(L, \phi) = L$. (note, this is the same as Axiom A.4.)

Axiom D.2: For all $K \subset T \cup S$, $d(\phi, K) = \phi$.

Axiom D.3: For all $L \subset T$ and $K, K' \subset S \cup T$,

$$d(L, K \cup K') \equiv d(d(L, K), K') .$$

Axiom D.4: For all $L, K \subset S$, and $J \subset T \cup S$,

$$d(L, K) = d(K, J) = \phi$$

implies $d(L, J) = \phi$.

These axioms can be intuitively interpreted as follows: Axiom D. 1 says that "subtracting" nothing (i. e., ϕ) from a subset $L \subset T$ gives us L (so ϕ serves as a zero). Axiom D. 2 says that subtracting something from nothing still results in nothing. Axiom D. 3 says (subject to our earlier interpretation of $\equiv$) that we can break up the taking of the difference into a series of differences. Axiom D. 4 says that the covering relation is transitive; i. e., it says that if K covers L and J covers K, then J covers L.

Given these axioms, we can now prove that the relation $\equiv$ has the desired properties.

Proposition 3. 1: If Axiom D. 4 holds and for every $L \subset T$ there exists L' such that $L \equiv L'$, then the relation $\equiv$ is an equivalence relation.

Proof: We already know that $\equiv$ is symmetric from its definition. That it is transitive follows easily from D. 4 for if $L_1, L_2, L_3 \subset T$ and $L_1 \equiv L_2$, $L_2 \equiv L_3$, then we have

$$d(L_1, L_2) = d(L_2, L_1) = d(L_2, L_3) = d(L_3, L_2) = \phi$$

so, by D. 4, $d(L_1, L_3) = d(L_3, L_1) = \phi$, i. e., $L_1 \equiv L_3$. Finally, from the assumption that for each $L \subset T$ there exists $L' \subset S$ such that $L' \equiv L$, we have $d(L, L') = d(L', L) = \phi$, so, by D. 4, $d(L, L) = \phi$, i. e., $L \equiv L$. Q. E. D.

Corollary 3. 2: If L, $L' \subset T$, $K \subset S$ and $L \equiv L'$, then K covers L (i. e., $d(L, K) = \phi$) implies K covers L'.

Proof: This is an immediate consequence of the transitivity of $\equiv$. Q. E. D.

The following simple result is also important.

Proposition 3. 3: If D. 1 and D. 2 hold and if $L \subset T$, then $L \equiv \phi$ if and only if $L = \phi$.

Proof: If $L = \phi$ then $d(L, \phi) = L = \phi$ by D. 1, and $d(\phi, L) = \phi$, by D. 2, hence $L \equiv \phi$ by definition.

Conversely, if $L \equiv \phi$, then by the definition of $\equiv$, $d(L, \phi) = \phi$, but by D. 1, $d(L, \phi) = L$, thus $L = \phi$. Q. E. D.

Theorem 3. 4: The D-axioms imply Axiom A. 1; indeed, if $k \in K$ and $d(L, I \cup (K-k)) \neq \phi$ then k is in every (K/I)-cover of L.

Proof: Say there exists a (K/I)-cover C of L which does not contain k. Then, perforce, $C \subset K - k$. Let $J = I \cup (K-k)$, then

$$
\begin{aligned}
d(L, I \cup (K-k)) &= d(L, J) \\
&= d(L, C \cup (J-C) \\
&\equiv d(d(L, C), J-C) && \text{by D. 3} \\
&\equiv d(\phi, J-C) && \text{by choice of } C \\
&\equiv \phi && \text{by D. 2.}
\end{aligned}
$$

But, by assumption, $d(L, I \cup (K-k)) \neq \phi$, so we have a contradiction unless no such (K/I)-cover C exists. Q. E. D.

Theorem 3.5: The D-axioms imply Axioms A.2; that is, if M is a minimal (K/I)-cover of L and $k \in M - I$, and Q is a minimal $((K-k)/(I \cup k))$-cover of $d(L, k)$, then the D-axioms imply that Q is a minimal (K/I)-cover of L.

Proof: We see first that Q is a (K/I)-cover of L since

ϕ =	$d(d(L, k), Q)$	by choice of Q
$\equiv$	$d(L, Q \cup k)$	by D.3
=	$d(L, Q)$	since, by definition, $k \in Q$.

But also we see that M is a $((K-k)/(I \cup k))$-cover of $d(L, k)$ since $I \cup k \subset M$ and

$d(d(L, k), M)$

$\equiv$	$d(L, M \cup k)$	by D.3
$\equiv$	$d(L, M) = \phi$	by definition of M, $k \in M$.

Thus, the fact that Q is a minimal $((K-k)/(I \cup k))$-cover

of $d(L, k)$ implies that $c^*(Q) \leq c^*(M)$ and so, since Q is a (K/I)-cover of L of cost less-than-or-equal to that of a minimal (K/I)-cover of L, we see that Q must also be a minimal (K/I)-cover of L. Q. E. D.

We will need the following lemma:

Lemma 3.6: If C is (K/I)-cover of L, $X \subset C - I$ and $Y \subset K - I$ such that $X \cap Y = \phi$ and

$$d(d(L, d(L, I \cup X)), I \cup Y) = \phi$$

then $C' = (C-X) \cup Y$ is also a (K/I)-cover of L.

Proof: Let $N = C - X$, then

$$d(L, N \cup X) = d(L, C) = \phi .$$

But, by D. 3,

$$d(L, N \cup X) = d(d(L, I \cup X), N - I) . \qquad (1)$$

Now, by assumption,

$$\begin{aligned} \phi &= d(d(L, d(L, I \cup X)), I \cup Y) \\ &\equiv d(L, I \cup Y \cup d(L, I \cup X)) \qquad \text{by D. 3} \\ &\equiv d(d(L, I \cup Y), d(L, I \cup X)) \qquad \text{by D. 3 .} \end{aligned}$$

Then, combining this with (1), using D. 4, we have

$$\begin{aligned} \phi &= d(d(L, I \cup Y), N - I) \\ &\equiv d(L, I \cup Y \cup (N-I)) \qquad \text{by D. 3} \\ &\equiv d(L, N \cup Y) \\ &= d(L, (C-X) \cup Y) = d(L, C') . \end{aligned}$$

Hence, C' is a (K/I)-cover of L. Q. E. D.

Axioms on Cost

Axiom C. 1: For all K, K', $I \subset S$, with $K \cap K' = \phi$,

$$c(I, K \cup K') = c(I, K) + c(I \cup K, K') .$$

Axiom C. 2: For all $K, K', I \subset S$ with $K \cap K' = \phi$,

$$c(I, K) + c(I, K') \geq c(I, K \cup K') .$$

Axiom C. 3: Given $k \in K \subset S$ and $X, Y \subset K - \Pi_K(k)$, then for all $Z \subset \Pi_K(k)$,

$$c(X,Z) = c(Y, Z) = c(\phi, Z) .$$

Axiom C. 4: For all $I, K \subset S$, if $k, k' \in J = K - I$ with $\Pi_J(k) \neq \Pi_J(k')$ and such that $d(d(L, d(L, I \cup \Pi_J(k))), I \cup \Pi_J(k')) = \phi$ and $c(I, \Pi_J(k')) \leq c(I, \Pi_J(k))$, then for every $X \subset \Pi_J(k)$ there exists $Y \subset \Pi_J(k')$ such that $d(d(L, d(L, I \cup X)), I \cup Y) = \phi$ and $c(I, Y) \leq c(I,X)$.

These axioms may be interpreted as follows: Axiom C. 1 says, in effect, that the cost of a subset of S (with respect to $I \subset S$) does not depend on the order in which we choose the subset. Axiom C. 2 says that the cost of a subset of S is not greater than the sum of the costs of its elements. (Note that this assumption restricts us in that it forbids cost functions that contain

a penalty for, say, fan-out over a certain amount.) Axiom C.3 says that all cost reductions (situations where $c(M, k) \leq c(\phi, k)$, for some $M \subset S$) take place with respect to the blocks of the partition Π. The final axiom, C.4, is the most complex. The idea here is that if $\Pi_J(k')$ will cover as much as $\Pi_J(k)$ and at no greater cost, then for each subset X of $\Pi_J(k)$ we can find a subset Y of $\Pi_J(k')$ which covers everything covered by X and which costs no more than X, (all this, of course, being with respect to the given I and L).

Theorem 3.7: The D and C axioms together imply Axiom A.3; that is, if $I, K \subset S$, $L \subset T$, and $k, k' \in J = K - I$, with $\Pi_J(k) \neq \Pi_J(k')$,

$$d(d(L, d(L, I \cup \Pi_J(k))), I \cup \Pi_J(k')) = \phi$$

and $c(I, \Pi_J(k')) \leq c(I, \Pi_J(k))$, then there exists a minimal $((K - \Pi_J(k))/I)$-cover of L and every $((K - \Pi_J(k))/I)$-cover of L is a (K/I)-cover of L.

Proof: Clearly it suffices to show that there is at least one $((K - \Pi_J(k))/I)$-cover of L which is a minimal (K/I)-cover. To show that such a cover exists, we will show that under the conditions of the theorem, we can transform any (K/I)-cover C of L into a corresponding $((K - \Pi_J(k))/I)$-cover Q of L with $c^*(Q) \leq c^*(C)$.

Let C be any (fixed) (K/I)-cover of L. Let $X = C \cap \Pi_J(k)$. By C.4 we know there exists $Y \subset \Pi_J(k')$ such that

$$d(d(L, d(L, I \cup X)), I \cup Y) = \phi$$

and $c(I, Y) \leq c(I, X)$. Now take $Q = (C - X) \cup Y$.

By Lemma 3.6 we know that Q is a $((K - \Pi_J(k))/I)$-cover of L; it remains to show that $c^*(Q) \leq c^*(C)$.

Now, by C.1

$$c^*(Q) = c(\phi, I) + c(I, Y) + c(I \cup Y, C - (I \cup X)),$$

and $$c^*(C) = c(\phi, I) + c(I, X) + c(I \cup X, C - (I \cup X)).$$

Now $c(I, Y) \leq c(I,X)$ by the above. Thus it remains only to compare the final terms. Let $W = (C - (I \cup X)) \cap \Pi_J(k')$, and let $U = (C - (I \cup X)) - \Pi_J(k')$, then

$$c(I \cup Y,\ C - (I \cup X)) = c(I \cup Y, U) + c(I \cup Y \cup U, W) \qquad (1)$$

and

$$c(I \cup X,\ C - (I \cup X))$$

$$= \quad c(I \cup X, U) + c(I \cup X \cup U, W) \qquad \text{by C. 1}$$

$$= \quad c(I \cup X, U) + c(I \cup U, W) \qquad \text{by C. 3. (2)}$$

Now, by C. 3, $c(I \cup X, U) = c(I \cup Y, U)$; thus it remains only to compare the final terms of (1) and (2). But

$$c(I \cup U,\ Y \cup W) = c(I \cup U, Y) + c(I \cup U \cup Y, W) \quad \text{by C. 1}$$

and $\quad c(I \cup U,\ Y \cup W) \leq c(I \cup U, Y) + c(I \cup U, W) \quad$ by C. 2,

which gives us

$$c(I \cup U \cup Y, W) \leq c(I \cup U, W)$$

just as desired in order to make (2) less-than-or-equal (1). Q. E. D.

To end this section we point out two simple refinements which can be made in the extraction algorithm.

In the extraction algorithm as given in Section 2, step 1 is the main means for rapidly reducing the size of K. Basically, the rule in step 1 is to throw away all elements of K which cover some part of L which can be covered more cheaply by other elements of K. Our purpose here is to prove the intuitively obvious extension of this rule to the effect that if an element of K covers nothing in L (not already covered by I) then it can be thrown out regardless of its cost.

Proposition 3.8. If $k \in K - I$ but I covers k with respect to L, then there exists a minimal (K/I)-cover M of L with $k \notin M$.

Proof: Say Q is a minimal (K/I)-cover of L and $k \in Q$. Let $Q = I \cup R \cup k$ (I, R, k disjoint). Since I covers k with respect to L, this means

$$d(d(L, d(L, k)), I) = \phi . \tag{1}$$

But since Q is a (K/I)-cover of L , we have

$$\phi = d(L,\ I \cup R \cup k)$$

$$= d(d(L, k),\ I \cup R) \qquad \text{by D. 3.} \qquad (2)$$

But from (1) we have

$$\phi = d(d(L, d(L, k)), I) \qquad \text{D. 3}$$

$$= d(L, I \cup d(L, k)) \qquad \text{D. 3}$$

$$= d(d(L, I),\ d(L, k)) = \phi \qquad (3)$$

So, combining (2) and (3), using D. 4, we get

$$\phi = d(d(L, I),\ I \cup R)$$

$$= d(L,\ I \cup I \cup R)$$

$$= d(L, M)$$

so M is a (K/I)-cover of L , but

$$C^*(Q) = C(\phi, M) + C(M, k) \qquad \text{by C. 1}$$

$$\geq C(\phi, M) \qquad \text{by definition C .}$$

Hence, M must be a minimal (K/I)-cover of L . Q. E. D.

In the extraction algorithm, as given in Section 2, step 4 picks out only one element at a time satisfying the conditions given in step 3. However, the one-at-a-time instruction is not central to the axiom and, indeed, we have:

Proposition 3. 9: If the D-axioms hold, then we can replace step 4 of the extraction algorithm with 4' . Let E be the set of all such elements (i. e., $k \in K$, $d(L, I \cup (K-k)) \neq \phi$) . If $d(L, I \cup E) = \phi$ then take

$$M(L, K/I) = I \cup E$$

and stop; otherwise, take

$$M(L; K/I) = M(d(L, E), (K-E)/(I \cup E)) .$$

Proof: Inspection of the proof of Theorem 3. 5 will show that it can be directly generalized to read:

"If M is a minimal (K/I)-cover of L and $X \subset M - I$, and Q is a minimal $((K-X)/(I \cup X))$-cover of $d(L, X)$, then the D-axioms imply that Q is a minimal (K/I)-cover of L."

(The proof is identical to that of 3.5 except that X replaces k throughout).

The desired result now follows directly from 3.4 and the above modification of 3.5, for by 3.4 we know that E must be a subset of every minimal (K/I)-cover and from the above modification of 3.5 we know that (by taking $X = E$) we get that

$$M(d(L, E), (K-E)/(I \cup E))$$

is thus a minimal (K/I)-cover of L. Q. E. D.

4. EXAMPLES

We will now give two, rather sketchy, examples of the above extraction algorithm, the first being single output, two-level AND-OR circuit minimization, the second being multiple-output, two-level AND-OR circuit minimization. The examples are presented without a proof of their validity (i. e., that the given d, and c satisfy the axioms). However, the validity follows easily from the material in [R-1] (especially if one considers it in terms of the geometric interpretation of the #-algorithm).

4.1 Single Output Case

It has been shown by Roth that the problem of designing minimal cost two-level AND-OR circuits can be reduced to a cubical covering problem [R-1] [R-2]. This problem is exactly of the type to which the extraction algorithm given in this paper can be applied. For a problem with n input variables we get that

S and T are the set of all faces of the n-cube;

L is a cover of the set of vertices of the n-cube which correspond to those conditions for which the circuit is to be ON ;

K is a cocycle cover of L (or, if there is a set D of DON'T-CARE vertices, then K is a cocycle cover of $L \cup D$);

d is the sharp-product (#-product) for covers [R-1];

c the cost, can be chosen in many ways, the most common being to make the cost of a k-cube being (n-k)+1 (this corresponds to the cost of a circuit being directly proportional to the number of inputs to logical blocks);

$\Pi = K$.

4.2 Multiple Output Case

A more interesting covering problem, and one with a nontrivial partition Π, arises in the design of multiple output two-level AND-OR circuits.

In the case that there are n inputs and m outputs we have that

$$S = T = \{0, 1, X\}^n \times \{1, \ldots, m\} .$$

Let a typical element of S or T be denoted $\langle q, i \rangle$ $(q \in \{0, 1, X\}^n,\ i \in \{1, \ldots, m\})$.

$L = \bigcup_{i=1}^{m} L_i$ where, for $i = 1, \ldots, m$, L_i is a set

$$L_i = \{\langle q_1^i, i\rangle, \ldots, \langle q_{p(i)}^i, i\rangle\}$$

such that $C_i = \{q_1^i, \ldots, q_{p(i)}^i\}$ constitutes a cover of the on-array of the i^{th} output.

K is the smallest set containing the cocycle cover of each set C_i (see above) and such that if $\langle q, i\rangle$, $\langle q', j\rangle \in K$ with $i \neq j$ and $q \sqcap q' \neq \phi$ then $\langle q \sqcap q', i\rangle$ and $\langle q \sqcap q', j\rangle$ are both in K .

d is the evident extension of the sharp-product for covers which arises from the rule

$$d(\langle q,i\rangle , \langle q',j\rangle) = \begin{cases} \phi & \text{if } q\#q' = \phi , \ i = j \\ \langle q\#q', i\rangle & \text{if } i = j, \text{ and } q\#q' \neq \phi \\ \langle q, i\rangle & \text{if } i \neq j . \end{cases}$$

Π is the partition which arises from the equivalence relation $\simeq$ (on K) such that

$$\langle q, i\rangle \simeq \langle q', j\rangle$$

if and only if $q = q'$.

c is such that for each $\langle q, i\rangle \in K$, if q is a k-cube, then

$$c(\phi, \langle q, i\rangle) = (n-k) + 1$$

and, if $I \cap \Pi_J(\langle q, i\rangle) \neq \phi$ (and $\langle q, i\rangle \notin I$), then

$$c(I, \langle q, i\rangle) = 1 .$$

This corresponds to the cost of a circuit again being directly proportional to the number of inputs to logical blocks. The first time we use $\langle q, i\rangle$ we have to pay for its inputs (n-k of them), and its input to the OR of the i^{th} output; but after that (since we already have the block for q), we only have to pay for its input to the output OR .

This multiple output algorithm is closely akin to the multiple output algorithm developed under this contract by Paul Roth. His algorithm, however, introduces a far more compact and convenient manner for handling the sets $\Pi_J(\langle q, i \rangle)$ (i. e., in his approach, each such set is a singular cube).

REFERENCES

[R-1] J. Paul Roth, "Algebraic topological methods for the synthesis of switching systems," Trans. American Mathematical Society, 88 (July 1958) 301-326; also ECP 56-02, Institute for Advanced Study, Princeton University (April 1956).

[R-2] J. Paul Roth, "Algebraic topological methods for the synthesis of switching systems--II," Annals of the Computation Laboratory, Harvard, 29 (1959) 57-73.

[R-W] J. Paul Roth and Eric G. Wagner, "Algebraic topological methods in the synthesis of switching systems--III, Minimization of nonsingular Boolean trees," IBM Journal of Research and Development, 4 (1959).

[E-R-W] A. Ewing, J. P. Roth, E. G. Wagner, "Algorithms for logical design," Communication and Electronics (September 1961) 1-8.

TOWARD A FORMAL THEORY OF SWITCHING CIRCUITS*

by

Eric G. Wagner
J. Paul Roth

IBM Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: This paper is a report on the beginnings of a formal theory of switching circuits. A formal system, the calculus of α-objects, is introduced which provides a uniform means for defining the mathematical objects, operations, and algorithms of switching theory in a strictly precise manner. This calculus is then employed to develop formal definitions of such objects as "logical components" and "combinational circuits" and to present an algorithm for the analysis of combinational circuits. These definitions and algorithms are presented without proof, but with motivation.

* This research was supported in part by J. P. L. Contract #951-538.

0. INTRODUCTION

0.1

Roth's Cubical Notation and calculus of cubes were originally developed [R-1] for application in the minimization of single output, two-level AND-OR switching circuits. In subsequent papers [R-W-1] [R-K], Roth and others applied this theory to synthesizing other forms of combinational circuits but without introducing a direct cubical notation for such circuits. However, in 1967, Roth introduced an informal cubical notation for representing arbitrary combinational circuits [R-2], and in 1968 the authors developed a semiformal cubical notation for multiple-output, two-level AND-OR circuits [R-W-2]. This variety of informal, semiformal and formal notations led the authors to consider the possibility of producing some kind of formal mathematical framework which would encompass all these diverse notations and which would permit the development of a general calculus for their manipulation. The hope was, and is, that the development of such a system would lead to rigorous and effective techniques for the analysis and synthesis of circuits. This

paper presents our results to date on the development of such a general framework.

In our initial attempts to produce a suitable framework, we tried to develop a suitable set of axioms which concerned the structures in which we were interested. None of these axiomatic approaches was particularly successful, for we found that we wanted to be able to deal with a great variety of structures and that further research would result in the discovery of even more structures. What we needed was a rather general approach that would allow us to build up "any" type of mathematical structure in a uniform manner. We turned then to the search for such a general approach and the result was the calculus of α-objects given in the first section of this paper. The calculus of α-objects is essentially a means (a formal procedure) for building up recursive definitions of classes of strings of symbols. What we present here is a specific such calculus which builds up classes of strings corresponding to the entities (truth tables, components, circuits, etc.) which make up the subject matter of switching theory and which, at the same time, provides means for

defining all the necessary operations and algorithms on, and relationships between, such entities.

The calculus of α-objects, presented in the first section of this paper, is intended to be more than just a notation, or language, for writing down the definitions of the entities and operations in which we are interested. A central idea here is to make the definitions, as well as the things they serve to define, into well defined mathematical objects. The idea of formalizing definitions is, of course, not new; our approach here bears at least a superficial resemblance to Smullyan's formal systems ['S']. However, where Smullyan's interest was primarily in developing a theory of formal systems, our interest is directed more to developing a calculus of definitions which is "application oriented." That is, we are not particularly interested in an alternative development of recursive function theory. Rather, we wish to develop powerful means for writing rigorous definitions of new structures and for proving results concerning them. In this paper, in keeping with its early position in the development, the emphasis is on the application to setting up the basic definitions for a formal

switching theory rather than in using the calculus for the development of theorems concerning this theory.

Using the calculus of α-objects, as we develop it here, we rapidly reach a point at which fairly complex definitions and algorithms can be quite easily written down in a completely rigorous manner. The penalty we pay for this convenience is that we start from a formalism that is, at best, difficult for the uninitiated to relate to his prior knowledge of switching theory or Roth's calculus of cubes. For this reason, we give the second part of this introduction over to an informal presentation of a version of Roth's informal notation with examples, and at the end of the introduction, we present a somewhat informal overview of our new notation using the same examples.

In Section 1 of this paper we present the calculus of α-objects. That is, we present our formal system for defining classes of strings. The material in this section, except for the examples, is presented without reference to our intended switching theory applications. In Section 2 of the paper, we employ the α-object calculus to develop a selection of the basic definitions needed for a rigorous

switching theory. While the material in Section 2 goes as far as to define components, circuits, and presents a definition-algorithm for the analysis of circuits, it is preliminary in nature. We anticipate that further study will result in both a shorter and a more powerful set of basic definitions.

0.2

Informally, we think of a combinational circuit as being a network of (logical) components with no feedback; that is, there is no signal path from an output of a component back to one of its inputs. Each component is, in turn, a "black box" with n inputs and m outputs which accepts binary input signals and responds by putting out binary output signals. An example of (the block diagram of) such a circuit is shown in Figure 1. In a network of components, those component inputs, which are not fed by the outputs of any other component, are called the primary inputs of the circuit; and those component outputs, which do not feed the inputs of any other components, are called primary outputs. We assume that each line in a circuit

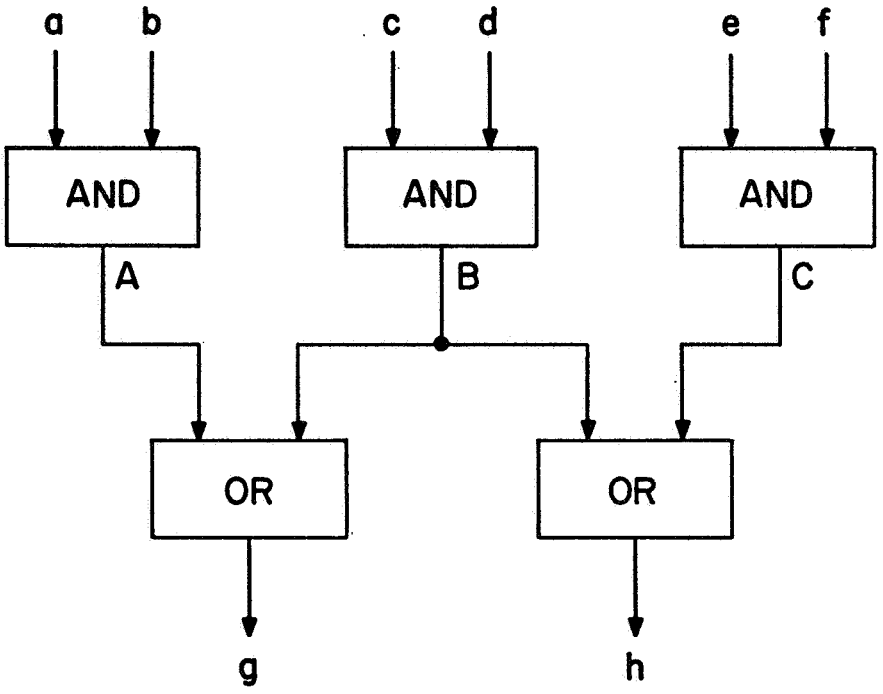


Figure 1 Example of a block diagram of a circuit.

has a label associated with it (a name or number); however, certain primary inputs may have the same label (may be identified as being fed by a common source of input signals) and we will also give the same label to (identify) all lines emanating from any given component output.

Now, any component or circuit realizes some binary function; that is, the relationship between its input and output signals is a binary function. This function can, of course, be represented by a table of 1's and 0's. However, it is much more convenient to represent it by a table of 1's, 0's, and X's, where the X's are used, as explained below, to reduce the size of the table. Such a table is called an Input-Output (or Truth) Table. To help explain this informal notation, we present in Figure 2 the input-output table for the circuit given in Figure 1. A 1 or 0 in a row of the table means that the signal on the corresponding line (input or output) is a 1 or 0, respectively. The X's have different meanings depending on whether they are in the input (left) or output (right) side of the table. In the input part, an X means that the

output does not depend on whether that input is a 1 or 0 (given that the other input lines are 1 or 0 as indicated). Thus, for example, the first line of the table in Figure 2 can be viewed as an abbreviation for the four lines

a	b	c	d	e	f	g	h
1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1
1	1	1	1	0	1	1	1
1	1	1	1	0	0	1	1

The X's in the output part of a row, on the other hand, mean that that row does not specify what the output is on the corresponding output line for the input signal combination specified in the input part of the row. Thus, for example, the fourth row of the table in Figure 2 says that output line g will have a 1 on it if input lines a, b, and c have respective input signals 1, 1 and 0, but it does not say anything about what we should expect on output line h for these input signals. The advantage of

a	b	c	d	e	f	g	h
1	1	1	1	X	X	1	1
X	0	1	1	X	X	1	1
0	X	1	1	X	X	1	1
1	1	0	X	X	X	1	X
1	1	X	0	X	X	1	X
X	X	X	X	1	1	X	1
0	X	0	X	X	X	0	X
0	X	X	0	X	X	0	X
X	0	0	X	X	X	0	X
X	0	X	0	X	X	0	X
X	X	0	X	0	X	X	0
X	X	0	X	X	0	X	0
X	X	X	0	0	X	X	0
X	X	X	0	X	0	X	0

Figure 2 Input-Output (or Truth) Table for circuit shown in Figure 1. (using X's in both inputs and outputs).

using the X's is, of course, that it often allows for a much shorter table; indeed, without the use of the X's, the input-output table would have to have 64 rather than 14 rows (indeed, with a more judicious use of X's, it is possible to get the table down to only 11 rows).

Now while the above table gives us the function relating the input and output signals of a circuit, it does not tell us anything about the structure of the circuits; that is, Figure 2 gives us a function, but it does not show us (as does the block diagram in Figure 1) that it arises from a circuit with three AND's and two OR's. To do this, to represent a circuit in a tabular rather than pictorial manner, we can use another form of table, also due to Roth [R-2]. The basic idea, as shown in Figure 3, is to form a "matrix" or table-of-tables which has columns for each input, output and intermediate line of the circuit and in which each subtable is a description of one of the components in the circuit. From such a table one can readily construct the corresponding block diagram. Now while such a table reduces block diagrams to a standard form, it is still not a "mathematical object" in the sense

a	b	c	d	e	f	A	B	C	g	h
1	1					1				
0	X					0				
X	0					0				
		1	1				1			
		0	X				0			
		X	0				0			
				1	1			1		
				0	X			0		
				X	0			0		
						1	X		1	
						X	1		1	
						0	0		0	
							1	X		1
							X	1		1
							0	0		0

Figure 3 "Matrix" representation of circuit shown in Figure 1.

that we can manipulate it in a rigorous manner. However, it was the consideration of just such circuit-representing "matrices" that originally prompted this research.

One of the aspects of such a circuit-representing "matrix" which stimulated research is that, given such a "matrix," it is not particularly difficult to produce from it a table which gives an "analysis" of the corresponding circuit. That is, one can produce a table, such as that in Figure 4, which shows the various combinations of signals which can appear on the lines of the circuit. Note that in Figure 4 we have again used X's. Here they mean (as in the input part of Figure 2) that the corresponding line can have either a 1 or 0 on it when the other lines are as indicated. (Again, the use of X's reduces the size of the table, in this case from 64 to 27 lines.)

We have now introduced three kinds of tabular representations of circuits. It is clear that there must be definite relationships between the different types of tables and that these relationships are of an essentially mathematical nature. However, since the different types

a	b	c	d	e	f	A	B	C	g	h
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	X	1	1	0	1	1
1	1	1	1	X	0	1	1	0	1	1
1	1	0	X	1	1	1	0	1	1	1
1	1	0	X	0	X	1	0	0	1	0
1	1	0	X	X	0	1	0	0	1	0
1	1	X	0	1	1	1	0	1	1	1
1	1	X	0	0	X	1	0	0	1	0
1	1	X	0	X	0	1	0	0	1	0
0	X	1	1	1	1	0	1	1	1	1
0	X	1	1	0	X	0	1	0	1	1
0	X	1	1	X	0	0	1	0	1	1
0	X	0	X	1	1	0	0	1	0	1
0	X	0	X	0	X	0	0	0	0	0
0	X	0	X	X	0	0	0	0	0	0
0	X	X	0	1	1	0	0	1	0	1
0	X	X	0	0	X	0	0	0	0	0
0	X	X	0	X	0	0	0	0	0	0
X	0	1	1	1	1	0	1	1	1	1
X	0	1	1	0	X	0	1	0	1	1
X	0	1	1	X	0	0	1	0	1	1
X	0	0	X	1	1	0	0	1	0	1
X	0	0	X	0	X	0	0	0	0	0
X	0	0	X	X	0	0	0	0	0	0
X	0	X	0	1	1	0	0	1	0	1
X	0	X	0	0	X	0	0	0	0	0
X	0	X	0	X	0	0	0	0	0	0

Figure 4 Analysis table of circuit shown in Figure 1.

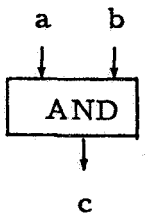
of tables are "informal objects," it is not possible to build up directly a calculus for their manipulation or which displays these interrelationships. The reason for this is that while we have examples of the different kinds of tables, we do not have the precise definitions which are necessary to make mathematical manipulation possible. We need to be able to describe, or define, the tables in such a way that we can decide precisely when an "arbitrary table" of 1's, 0's, or X's is one of the kinds of tables we are interested in. We need precise means by which to specify the parts of a table; we need to define basic operations on tables and parts of tables. Our tool for doing these things will be the α-object calculus.

0.3

The problem of formalizing the above informal tabular notations is largely one of replacing the tables with a more readily describable and manipulatable form. To do this we have taken the route of reducing everything to strings of symbols. The actual set of symbols which we use in the formal development is the set $\{0, 1, X, \bar{o}, \langle, \rangle\}$;

however, in this introduction, we shall use some additional symbols in order 1) to make it easier to relate to the tabular notation, and 2) to avoid long strings. We proceed by example.

Consider the following AND-circuit and its input-output table.



a	b	c
1	1	1
0	X	0
X	0	0

The basic idea of our notation is to represent the table of such an AND by a string of the following kind:

$$\langle(\langle a1\rangle\langle b1\rangle \mid \langle c1\rangle)(\langle a0\rangle \mid \langle c0\rangle)(\langle b0\rangle \mid \langle c0\rangle)\rangle \ .$$

(In the actual formal notation, the "labels" a, b, and c would be strings of 1's, 0's, and X's in angle brackets ($\langle$ and $\rangle$) and the parentheses and vertical slashes would be replaced by appropriate configurations

of brackets.)

Note that in this notation, a row of the matrix becomes a string (delineated by parentheses) with the X-entries omitted, and with the inputs and outputs separated by a vertical slash. Then the component is represented by a string of such row-representing-strings delineated angle brackets. If we want to represent the input-output-(or truth)-table of a circuit, we do it in just the same way. Thus table 2 is represented by the string:

⟨(⟨a1⟩⟨b1⟩⟨c1⟩⟨d1⟩ | ⟨g1⟩⟨h1⟩)
(⟨b0⟩⟨c1⟩⟨d1⟩ | ⟨g1⟩⟨h1⟩)
(⟨a0⟩⟨c1⟩⟨d1⟩ | ⟨g1⟩⟨h1⟩)
(⟨a1⟩⟨b1⟩⟨c0⟩ | ⟨g1⟩)
(⟨e1⟩⟨f1⟩ | ⟨h1⟩)
(⟨a0⟩⟨c0⟩ | ⟨g0⟩)
(⟨a0⟩⟨d0⟩ | ⟨g0⟩)
(⟨b0⟩⟨c0⟩ | ⟨g0⟩)
(⟨b0⟩⟨d0⟩ | ⟨g0⟩)
(⟨c0⟩⟨e0⟩ | ⟨h0⟩)
(⟨c0⟩⟨f0⟩ | ⟨h0⟩)
(⟨d0⟩⟨e0⟩ | ⟨h0⟩)
(⟨d0⟩⟨f0⟩ | ⟨h0⟩)⟩.

To represent a circuit (combinational network of components), it suffices to give a string which is an appropriate ordering of the strings representing the individual components. Thus, corresponding to Table 3, we have the string:

$$\langle\langle(\langle a1\rangle\langle b1\rangle \mid \langle A1\rangle)(\langle a0\rangle \mid \langle A0\rangle)(\langle b0\rangle \mid \langle A0\rangle)\rangle\langle(\langle c1\rangle\langle d1\rangle \mid \langle B1\rangle)$$
$$(\langle c0\rangle \mid \langle B0\rangle)(\langle d0\rangle \mid \langle B0\rangle)\rangle\langle(\langle e1\rangle\langle f1\rangle \mid \langle C1\rangle)(\langle e0\rangle \mid \langle C0\rangle)$$
$$(\langle f0\rangle \mid \langle C0\rangle)\rangle\langle(\langle A1\rangle \mid \langle g1\rangle)(\langle B1\rangle \mid \langle g1\rangle)(\langle A\rangle\langle B0\rangle \mid \langle g0\rangle)\rangle$$
$$\langle(\langle B1\rangle \mid \langle h1\rangle)(\langle C1\rangle \mid \langle h1\rangle)(\langle B0\rangle\langle C0\rangle \mid \langle h0\rangle)\rangle\rangle\ .$$

To represent the analysis-table of a circuit by a string, we proceed in a manner similar to the above. In that the notation is rather bulky, we will not write out the string corresponding to Table 4.

The natural question at this point is: What have we gained by going to such a string notation? The answer is that we shall be able to work with these strings in a rigorous manner. In particular, we shall be able to say just which strings correspond to possible components and circuits and we will be able to present precise rules for,

say, going from a string representing a circuit to one representing an analysis table for the same circuit.

1. THE α-OBJECT CALCULUS

1.0 Introductory remarks

In this section we develop the rudiments of the α-object calculus. What we present is a formal system for writing recursive definitions of strings (finite sequences) of the symbols 0, 1, X, $\bar{o}$, $\rangle$, and $\langle$. The system is formal in that it has a "grammar," or precise set of rules, which effectively define what we mean by a "definition" and there is an accompanying set of precise rules, the "semantics," which (albeit not necessarily effectively) determine the set of strings "defined" by a given "definition" or "string of definitions." Inasmuch as the formal aspects of the calculus play only a minor role in this preliminary paper, the reader can comprehend the material presented in Section 2 of this paper without appreciating the formal aspects of the definitions. That is, in this paper, one can view the α-object calculus as just a notation. We wish to point out, however, that even in this paper, the formal framework assures the completeness of the definitions in the sense that 1) we have no undefined terms floating around,

and 2) every definition defines a definite class of strings whether or not it is the one desired.

In Section 1.1 the first four pages are dedicated to defining our formal notion of a ("string of") "definition(s)." On the fifth page we finally get to the question of the "meaning" of a "definition;" that is, to the rules which determine the corresponding class of strings. While this manner of presentation is well justified mathematically, it makes it somewhat difficult for the reader to get any feeling for what is going on; thus we will close these introductory remarks with an informal description of the α-object calculus viewed as a notation for writing recursive definitions.

The basic idea of the α-Object Calculus as a notation is to provide a simultaneous means for defining and naming classes of strings on the alphabet 0, 1, X, $\bar{o}$, $\rangle$, and $\langle$. The "names" are important for they allow us to refer to a class of strings when we are defining further classes of strings, or when we have a recursive definition. It allows us to refer to a given class in building up its own definition (indeed this self-referral aspect

is the essence of a recursive definition). A simple example of such a definition would be the following definition of the class named, say, "STRINGS-OF-ONES."

A.1. The symbol "1" is in the class STRINGS-OF-ONES.

A.2. If A and B are strings in the class STRINGS-OF-ONES, then their concatenation AB is in the class STRINGS-OF-ONES.

A.3. No string is in the class STRINGS-OF-ONES unless its being so follows from A.1 and/or A.2.

As an example of the use of the class STRINGS-OF-ONES in a further definition, we might define a class called, say, BSOBPOBSOO (for "Bracketed Strings Of Bracketed Pairs Of Bracketed Strings of Ones").

B.1. If A and B are strings in the class of STRINGS-OF-ONES, then $\langle\langle\langle A\rangle\langle B\rangle\rangle\rangle$ is a string in the class BSOBPOBSOO.

B.2. If $\langle A\rangle$ and $\langle B\rangle$ are strings in the class

BSOBPOBSOO, then ⟨AB⟩ is a string in the class BSOBPOSBSOO.

B. 3. No string is in the class BSOBPOBSOO unless its being so follows from B. 1 and/or B. 2 and the definition of STRINGS-OF-ONES.

(Examples: ⟨⟨⟨111⟩⟨1111⟩⟩⟩ and ⟨⟨⟨11⟩⟨11111⟩⟩⟩ are in BSOBPOBSOO by B. 1 and the definition of STRINGS-OF-ONES; and ⟨⟨⟨111⟩⟨1111⟩⟩⟨⟨11⟩⟨11111⟩⟩⟩ is in BSOBPOBSOO by the above and B. 2.)

Viewed as a notation, the α-object calculus provides a notation for writing definitions of the above type in a uniform and condensed manner. There are four main notational conventions:

1. Given that we have defined or are defining a class of strings named, say α, and we have a string or symbols s standing for a string (such as A and B above), then

we write

$$\alpha[s]$$

as an abbreviation for "s is a string in the class α." Thus, for example, STRINGS-OF-ONES [1] means "the symbol 1 is in the class STRINGS-OF-ONES; and BSOBPOBSOO[⟨A⟩] means "the string ⟨A⟩, consisting of the string (denoted by the variable) A enclosed in brackets, is in the class BSOBPOBSOO."

2. A sequence $\alpha_1[s_1], \alpha_2[s_2], \ldots, \alpha_n[s_n]$ is read as a conjunction; i. e., the above would be read as "s_1 is a string in the class α_1, and s_2 is a string in the class α_2,... and s_n is a string in the class α_n."

3. We employ an arrow "→" to denote the "if... then" part of the sentences in a definition, and we enclose the whole abbreviated sentence in parentheses. Thus A. 2 is written

(STRINGS-OF-ONES [A], STRINGS-OF-

ONES [B] → STRINGS-OF-ONES [AB]).

The arrow is also used in abbreviating sentences such as A. 1 where there is no "if". Where there is no "if" nothing is written to the left of the arrow and thus A. 1 is abbreviated as

(→ STRINGS-OF-ONES [1]) .

4. Finally, sentences such as A. 3 and B. 3 are omitted. Thus, the above examples of definitions can be rewritten as:

(→ STRINGS-OF-ONES [1])

(STRINGS-OF-ONES [A], STRINGS-OF-ONES [B] → STRINGS-OF-ONES [AB]

(STRINGS-OF-ONES [A], STRINGS-OF-ONES [B] → BSOBPOBSOO [⟨⟨⟨A⟩⟨B⟩⟩⟩])

(BSOBPOBSOO [⟨A⟩], BSOBPOBSOO [⟨B⟩] → BSOBPOBSOO [⟨AB⟩]) .

Some further and important simplifications of the notation are given in Section 1. 3.

1.1 Formal presentation of the α-object calculus

Let the symbols 0, 1, X and $\bar{o}$ (zero, one, ex, and null) be called primitive-objects. We then define an object to be any string in the smallest set of strings satisfying the following definition:

1. all primitive-objects are objects;
2. if x and y are objects, then so is xy ;
3. if x is an object, then so is $\langle x \rangle$.

Given two objects θ_1, θ_2 we say they are equal and write $\theta_1 = \theta_2$ if and only if they are identical as strings on the alphabet $\{0, 1, X, \bar{o}, \langle, \rangle\}$.

We wish now to present a general method for defining various subclasses of the class of objects. Each definition will define a class (possibly empty) of objects with a given name. If the name is, say, α , we call the resulting objects (if any) α-objects.

Let p, q, r, s, t, u, v, w, x, y, z, p', q', ... be called variables. By a formal-term we mean

1. a variable or an object;

2. if $\underline{A}$ is a formal term, then $\langle \underline{A} \rangle$ is a formal term;

3. if $\underline{A}$ and $\underline{B}$ are formal terms, then so is $\underline{AB}$.

A formal-definition of α-objects in terms of $\beta_1, \ldots, \beta_n$-objects will consist of a (finite) sequence of formal-expressions of the form

$$(\gamma_1[X_1], \gamma_2[X_2], \ldots, \gamma_s[X_s] \rightarrow \alpha[Y_1], \ldots, \alpha[Y_t]) \tag{1}$$

where, for $i = 1, \ldots, s$, $\gamma_i \in \{\alpha, \beta_1, \ldots, \beta_n\}$, X_i is a formal term, and, for $k = 1, \ldots, t$, each Y_k is a formal-term in which no variable occurs that does not occur in at least one of the X_i. A formal-expression will be said to be a basis-expression if $\gamma_i \neq \alpha$ for $i = 1, \ldots, s$; otherwise, it will be called an inductive-expression. Each formal-definition will contain at least one basis-expression, and in a formal-definition, all the basis-expressions will come before the inductive-expressions. Let $D(\alpha, \{\beta_1, \ldots, \beta_n\})$ denote a formal-

definition of α-objects in terms of $\beta_1, \ldots, \beta_n$-objects.

Given a formal-expression such as (1) above, let $y_1, \ldots, y_n$ be the variables which appear in it. By an assignment of these variables, we mean a map $\underline{A}$ of $\{y_1, \ldots, y_n\}$ into the class of all objects; thus, $\underline{A}(y_i)$ is an object for all i. Uniformly substituting $\underline{A}(y_i)$ for y_i in X_j and Y_k for $i = 1, \ldots, n$, $j = 1, \ldots, s$, $k = 1, \ldots, t$. in the given formal expression, we get a new formal-expression

$$\gamma_1[\theta_1], \ldots, \gamma_s[\theta_s] \rightarrow \alpha[\theta'_1], \ldots, \alpha[\theta'_t]$$

which we call the $\underline{A}$-instance of the original formal-expression.

We shall also find it convenient to be able to speak of an instance of a formal term. For this we use the same notions as in the above paragraph. Clearly, every instance of a formal term is an object. If X is a formal term and θ is an object which is an instance of X, then we say that θ is of form X.

Given a formal-expression such as (1) above, we

call $\gamma_1[X_1], \ldots, \gamma_s[X_s]$ the left-side of the expression, and $\alpha[Y_1], \ldots, \alpha[Y_t]$ the right-side of the expression. The symbols $\alpha, \beta_1, \ldots, \beta_n$ occurring in an expression are called names.

By a definition-string we mean a (finite) sequence of formal-definitions such that:

1. No name occurs on the right-side of the formal-expressions in more than one formal definition;
2. the first definition in the sequence is
$$(\rightarrow P[0], P[1], P[X], P[\bar{o}]);$$
3. no name occurs on the left-side of a formal expression unless it has already occurred on the right-side of a formal-expression appearing earlier in the sequence.

It is easy to see that a definition-string will always be of the form

$$\underline{D} = D(P, \emptyset), D(\alpha_1, \{P\}), D(\alpha_2, \{P, \alpha_1\}, \ldots, D(\alpha_n, \{P, \alpha_1, \ldots, \alpha_{n-1}\}) .$$

Taking $P = \alpha_0$ we shall now give rules which associate with each α_i, $i = 0, 1, \ldots, n$ in $\underline{D}$, a unique (but possibly empty) class of objects which we then call the class of α_i-objects (with respect to $\underline{D}$). Let $B_0 = \emptyset$, and for $i > 0$, let $B_i = \{\alpha_0, \ldots, \alpha_{i-1}\}$.

Given $\underline{D}$, then for each α_i, the <u>class of α_i-objects</u> is defined to be the smallest class of objects such that:

1. If θ is an object and $D(\alpha_i, B_i)$ contains a formal expression

$$(\rightarrow \alpha_i[X_1], \ldots, \alpha_i[\theta], \ldots, \alpha_i[X_s])$$

then θ is an α_i-object;

2. if $D(\alpha_i, B_i)$ contains a formal-expression

$$(\gamma_1[X_1], \ldots, \gamma_s[X_s] \rightarrow \alpha_i[Y_1], \ldots, \alpha_i[Y_t])$$

and there exists an assignment $\underline{A}$ of the variables occurring in this expression such that the A-instance

$$(\gamma_1[\theta_1], \ldots, \gamma_s[\theta_s] \rightarrow \alpha_i[\theta_1'], \ldots, \alpha_i[\theta_t'])$$

has the property that θ_j is a γ_j-object for $j = 1, \ldots, s$ then, for $k = 1, \ldots, t$, θ_k' is an α_i-object.

Example:

$\mathscr{A}$ = (→P[0], P[1], P[X], P[$\bar{o}$])(P[x] →OBJECT [x])

(OBJECT [x], OBJECT [y] →OBJECT [⟨x⟩], OBJECT [xy])

(OBJECT [⟨⟨x⟩⟨y⟩⟩] → PAIR [⟨⟨x⟩⟨y⟩⟩]) .

Inspection will show that the class of OBJECT-objects defined by $\mathscr{A}$ is precisely the class of all objects and that the class of PAIR-objects is precisely the class of all objects of the form ⟨⟨x⟩⟨y⟩⟩ .

1.2 Functions and relations

"Conventional mathematics" deals with sets, relations and functions while we are dealing here only with objects (strings) and classes of objects. However, we will find it convenient, at least for expository purposes, to introduce notions analogous to the set theoretic notions of relation and function. These notions will be outside our theory in the sense that we will not define them by means of definition-strings.

We proceed as follows: First of all, we say that an object θ is a pair if it is of form ⟨⟨x⟩⟨y⟩⟩ (note that

we have already given a string-definition of PAIR-objects which agrees with this informal definition). We then say that the class of α-objects is a relation if every α-object is a pair. Finally, we say that the class of α-objects is a function if, 1) it is a relation, and 2) for all objects θ_1, θ_2, and θ_3, if $\langle\langle\theta_1\rangle\langle\theta_2\rangle\rangle$ and $\langle\langle\theta_1\rangle\langle\theta_3\rangle\rangle$ are α-objects, then $\theta_2 = \theta_3$.

It is worth noting that we may, of course, for each α interpret the class of α-objects as a set, or as a predicate. For example, we later define CIRCUIT-objects. The class of these objects is naturally viewed as a set, but, on the other hand, we can also view it as a predicate in the sense that if an object is a CIRCUIT-object (is in the class, satisfied the definition), then it has the property of being a circuit.

1.3 Informal simplifications of the α-object calculus

While it is perfectly possible to present our formal definitions purely within the formalism given above, it is clearly to the advantage of both the reader and the writer to introduce a number of conventions and short cuts

into the notation. We will now introduce two types of conventions: First, some which are purely notational or matters of format, and second, we introduce definition-schemas; that is, definitions with variables running over the set of class names as well as over objects.

As regards notation and format, each formal definition will be written as a "paragraph" headed by the name (or names) of the class of objects being defined, and then followed by the formal-expressions, one-to-a-line, which make up the formal definition. Furthermore, the name of the class being defined will be deleted from the right-side of the formal expression. Thus our Example 1 would appear as

P-objects

$(\rightarrow 0, 1, X, \bar{o})$

OBJECT-objects

$(P[x] \rightarrow x)$

$(OBJECT[x],\ OBJECT[y] \rightarrow \langle x \rangle,\ xy)$

PAIR-objects

$(OBJECT[\langle \langle x \rangle \langle y \rangle \rangle] \rightarrow \langle \langle x \rangle \langle y \rangle \rangle)$.

Since there will be many definitions involving pairs (or PAIR-objects), we shall often find it convenient for any objects θ_1 and θ_2 to write $\theta_1 | \theta_2$ or $(\theta_1 | \theta_2)$ for $\langle\langle\theta_1\rangle\langle\theta_2\rangle\rangle$.

The idea of a definition-schema is quite simple. All we mean is a formal-definition which contains variables standing for names as well as variables standing for objects. For example, there will be many situations when we will have defined some class, say the class of α-objects, and we will want to then define the class of "all bracketed strings of α-objects," i.e., the class of all strings of the form $\langle x_1 x_2 \ldots x_n\rangle$ where all the x_i are α-objects. Rather than write out a complete formal-definition each time this kind of situation arises for a new choice of α, we write out a general definition schema as follows:

B-STRING(α) or BSTR(α)

($\alpha[x] \rightarrow \langle x\rangle$)

(B-STRING[$\langle x\rangle$], B-STRING[$\langle y\rangle$] $\rightarrow \langle xy\rangle$).

Given this general definition, we can now define certain new classes of objects without writing out all the formal expressions. For example, we can denote the class of "all bracketed strings of bracketed strings of primitive objects" by B-STRING(B-STRING(P)) .

The use of such definition schemas not only cuts down on the amount that we have to write, but even more important, it helps to provide a unifying thread in a definition string by pointing out where different formal definitions have the same underlying form.

Definition schema will be particularly useful for dealing with relations and functions. To begin with, we can define the notion of the domain and image of a relation or function α with no trouble at all:

DOMAIN(α) or DOM(α)

$$(\alpha[x|y] \rightarrow x)$$

IMAGE(α) or IM(α)

$$(\alpha[x|y] \rightarrow y) .$$

(Note that DOM(α) and IM(α) are defined for any choice

of α though these classes will be empty if no α-objects are pairs; however, we will only use these definitions when we are dealing with relations or functions.)

The real use of definition schema in connection with relations and functions will be to extend or "lift" a relation or function from one domain to another. We now give the definition schema for several such "lifts". These lifts will prove very valuable later in the paper.

1.4 Three lifts for relations

Let α be a relation, that is, assume every α-object is of the form $x|y$. Then the following definition schemas define two new relations $\Sigma(\alpha)$ and $\Box(\alpha)$ and a new predicate $\Delta(\alpha)$. These definition schemas may, of course, be applied to any α whether or not it is a relation; however, we are only interested in the case where α is a relation and, indeed, in the interpretations given with the definition schema, we assume that α is a relation in which every α-object is of the form $\langle x\rangle|\langle y\rangle$ (i.e., α is a relation between objects of the form $\langle z\rangle$).

1. $\Sigma(\alpha)$

Interpretation: $\Sigma(\alpha)$ is the extension of α to the relation between DOM(α) and BSTR(IM(α)) such that $x|y_i$, $i = 1, \ldots, n$ are α-objects if and only if $x|\langle y_1 y_2 \ldots y_n\rangle$ is a $\Sigma(\alpha)$-object. However, if y_i, $i = 1, \ldots, n$, are not objects of form $\langle z\rangle$, then the "if" part of the above interpretation may not hold.

$\Sigma(\alpha)$

$(\alpha[x|y] \rightarrow x|\langle y\rangle)$

$(\Sigma(\alpha)[x|\langle y\rangle], \Sigma(\alpha)[x|\langle z\rangle] \rightarrow x|\langle yz\rangle)$.

2. $\Box(\alpha)$

Interpretation: $\Box(\alpha)$ is the extension of α to the relation between BSTR(DOM(α)) and BSTR(IM(α)) such that for DOM(α)-objects $a_1, \ldots, a_n$ and IM(α)-objects $b_1, \ldots, b_m$ we have $\langle a_1 a_2 \ldots a_n\rangle | \langle b_1 b_2 \ldots b_m\rangle$ is a $\Box(\alpha)$-object if and only if $a_i | b_j$ is an α-object for all i and j. Again this interpretation assumes that the a_i and b_i are always objects of the form $\langle x\rangle$.

$\Box(\alpha)$

$(\Sigma(\alpha)[x \mid y] \rightarrow \langle x \rangle \mid y)$

$(\Box(\alpha)[\langle x \rangle \mid y], \Box(\alpha)[\langle z \rangle \mid y] \rightarrow \langle xz \rangle \mid y)$

3. $\Delta(\alpha)$

Interpretation: $\Delta(\alpha)$ is the predicate consisting of all BSTR(DOM(α))-objects $\langle a_1 a_2 \ldots a_n \rangle$ such that $a_i \mid a_j$ is an α-object for $1 \leq i < j \leq n$.

(Again we assume each a_i is of the form $\langle z \rangle$).

$\Delta(\alpha)$

$(\alpha[x \mid x] \rightarrow \langle x \rangle)$

$(\Delta(\alpha)[\langle y \rangle], \alpha[x \mid x], \Sigma(\alpha)[x \mid \langle y \rangle] \rightarrow \langle xy \rangle)$.

1.5 Lifts for functions

We now introduce a number of lifts for functions which allow us to extend functions to more complex domains and images. These particular lifts will be of particular use in Section where they will permit us to define the analysis of a circuit in a very succinct and natural manner. As in the case of the lifts for relations, we will given an interpretation of these lifts which fits our

applications rather than the general case where α is arbitrary.

1. $F\Sigma 1(\alpha)$

Interpretation: If α is a function of a single variable, that is, if all α-objects are of the form $\langle x\rangle \mid \langle y\rangle$, then for $DOM(\alpha)$-objects $a_1, \ldots, a_n$ and $IM(\alpha)$-objects $b_1, \ldots, b_n$, we have $\langle a_1 a_2 \ldots a_n\rangle \mid \langle b_1 b_2 \ldots b_n\rangle$ is a $F\Sigma 1(\alpha)$-object if and only if $a_i \mid b_i$ is an α-object for $i = 1, \ldots, n$. Thus $a_1 a_2 \ldots a_n$ is a string of arguments for α and $b_1 b_2 \ldots b_n$ is the corresponding set of values.

$F\Sigma 1(\alpha)$

$(\alpha[x \mid y] \rightarrow \langle x\rangle \mid \langle y\rangle)$

$(F\Sigma 1(\alpha)[\langle x\rangle \mid \langle y\rangle], F\Sigma 1(\alpha)[\langle w\rangle \mid \langle z\rangle] \rightarrow \langle xw\rangle \mid \langle yz\rangle)$.

2. $F\Sigma 2(\alpha)$

Interpretation: Here α is assumed to be a function of two arguments; that is, each α-object is assumed to be of the form $\langle x\rangle\langle y\rangle \mid z$. The idea of $F\Sigma 2(\alpha)$ is that if all the second arguments (the $\langle y\rangle$'s) in

the α-objects are, say, β-objects, then we wish to replace the second argument by B-STRINGS of β-objects and get B-STRINGS as values.

$F\Sigma 2(\alpha)$

$(\alpha[\langle x\rangle\langle y\rangle \mid z] \rightarrow \langle x\rangle\langle\langle y\rangle\rangle \mid \langle z\rangle)$

$(F\Sigma 2(\alpha)[\langle x\rangle\langle y\rangle \mid \langle z\rangle], F\Sigma 2[\langle x\rangle\langle v\rangle \mid \langle w\rangle] \rightarrow \langle x\rangle\langle yv\rangle \mid \langle zw\rangle)$.

Thus, if we have $\alpha[\langle a\rangle\langle b_i\rangle \mid c_i]$ for $i = 1,\ldots, n$, then we get

$$F\Sigma 2(\alpha)[\langle a\rangle\langle\langle b_1\rangle\langle b_2\rangle \ldots \langle b_n\rangle\rangle \mid \langle c_1 c_2 \ldots c_n\rangle] .$$

3. $F\Box 2(\alpha)$

Interpretation: Given that we have α-objects $(a_i b_j \mid c_{ij})$ for $i = 1,\ldots, n$, $j = 1,\ldots, m$, then, we get

$$\langle a_1 \ldots a_n\rangle\langle b_1 \ldots b_m\rangle \mid \langle c_{11} c_{12} \ldots c_{nm}\rangle$$

is an $F\Box 2(\alpha)$ object. Thus, $F\Box 2$ lifts a function α to a function ranging over B-strings of the arguments of α .

$(F\Sigma 2(\alpha)[xy \mid z] \rightarrow \langle x\rangle y \mid z)$

$(F\Box 2(\alpha)[\langle x\rangle y \mid \langle z\rangle], F\Sigma 2[uy \mid \langle w\rangle] \rightarrow \langle xu\rangle y \mid \langle 2w\rangle)$.

4. FII(α)

Interpretation: If α is say a function of two arguments, say, $\alpha: B \times B \rightarrow B$ where B is the class of, say, β-objects, then for any B-STRING(β) $C = b_1 b_2 \ldots b_n$ we get

$$C \mid \alpha(\ldots\alpha(\alpha(x_1, x_2), x_3)\ldots, x_n)$$

is in FII(α).

FII(α)

$(\alpha[\langle x\rangle\langle y\rangle \mid z] \rightarrow \langle\langle x\rangle\langle y\rangle\rangle \mid z)$

$(\text{FII}(\alpha)[x \mid y], \alpha[\langle y\rangle\langle z\rangle \mid w] \rightarrow \langle xz\rangle \mid w)$

5. FP1(α)

Interpretation: Given a function α of one argument, this lift changes it to a function of one argument which takes PAIR objects as arguments and takes the value of α on the first object in the PAIR as its value.

FP1(α)

$(\alpha[x \mid y], \text{PAIR}[(x \mid z)] \rightarrow (x \mid z) \mid y)$

6. FP2(α)

Interpretation: Same as FP1 only it picks out the second object in the PAIR.

FP2

$(\alpha[x|y], \text{PAIR}[(z|x)] \rightarrow (z|x)|y)$

7. INVERT-ARG(α) or IA(α)

This definition schema does not give us a lift, but is useful in producing lifts. What it does is reverse the order of the arguments of a two-argument function α.

IA(α)

$(\alpha[\langle x\rangle\langle y\rangle | z] \rightarrow \langle y\rangle\langle x\rangle | z)$

2. BASIC DEFINITIONS OF SWITCHING THEORY

2.0

Our purpose in the remainder of this paper will be to write down a definition-string which will contain a significant subset of the structures, relations, and operations (functions), which we feel are basic to (combinational) switching theory. The goal of this particular section will be to define component and circuit and show how to analyze a circuit (in terms of our formalism). The first subsection defines some preliminary structures, the second subsection defines some basic operations and relations, the third subsection contains the definition of a partially specified component and circuit, the fourth subsection gives the definition of completely specified components and circuits, and the fifth subsection presents the analysis operation.

2.1 Some basic structures

In this subsection we define the basic structures of the subject. We start the definition-string. The classes of objects defined here are generally not of much

interest in themselves, but they provide a jumping off point for defining the classes of objects of interest in switching theory. In terms of the informal notation, what we do here is essentially to define the idea of arbitrary tables of 1's, 0's, and X's (i.e., tables not necessarily having anything to do with circuits) of the general form of the tables in Figures 2 and 3.

P-objects

$(\rightarrow 0, 1, X, \bar{o})$

IOX-objects

$(\rightarrow 0, 1, X)$

IO-objects

$(\rightarrow 0, 1)$

OBJECT-objects

$(P[x] \rightarrow x)$

$(OBJECT\ [x],\ OBJECT\ [y] \rightarrow \langle x\rangle, xy)$

ITEM-objects

$(OBJECT\ [\langle x\rangle] \rightarrow \langle x\rangle)$

PAIR-object

$(OBJECT[\langle\langle x\rangle\langle y\rangle\rangle] \rightarrow \langle\langle x\rangle\langle y\rangle\rangle)$

Notation: We shall often write $x|y$ or $(x|y)$ to abbreviate $\langle\langle x\rangle\langle y\rangle\rangle$.

We turn now to defining the basic concepts employed in this treatment of switching theory.

CUBE-object

(B-STRING(P)[x] → x)

(We will not make much use of CUBE-objects in this paper since we will be using LABELLED-CUBES (see below); however, the "traditional" calculus of cubes can be developed quite easily from this simple concept.)

LABEL-object

(B-STRING(IO)[x] → x)

(In the introduction, we used lower case letters for labels to formalize them by the above strings.)

PRIMITIVE-LABELLED-CUBE- or PL-CUBE-object

(LABEL[u] → ⟨u0⟩, ⟨u1⟩, ⟨uX⟩)

LABELLED-CUBE- or L-CUBE-object

(B-STRING(PL-CUBE)[x] → x)

(There is a direct relationship between LABELLED-CUBES and logical terms; e.g., if a, b, and c are labels, then the L-CUBE

⟨⟨a1⟩⟨b0⟩⟨cX⟩⟩

corresponds to the logical expression

$a\bar{b}(c \vee \bar{c}) = a\bar{b}$.)

LABELLED-COVER- or L-COVER-object

(B-STRING(L-CUBE)[x] → x)

(One can think of an L-COVER as a disjunction of the logical terms corresponding to L-CUBES.)

LABELLED-SINGULAR-CUBE- or LS-CUBE-object

(L-CUBE[x], L-CUBE[y] → ⟨xy⟩) .

2.2

In this subsection we introduce a number of relations and operations which will be employed in the next subsection to go from the general objects defined in the

preceding subsection to precise characterizations of the objects corresponding to components and combinational circuits. The most important (and most complex) operation (or function) introduced in this subsection is REDUCTION. Essentially this operation reduces a labelled cube down to its shortest logical equivalent by eliminating redundancies and contradictions. Using this operation we can easily define an operation INTERFACE corresponding exactly with Roth's interface operation [R-W-2]. REDUCTION will also be used in later definitions.

The first relation we define is the congruence relation on CUBES. Intuitively, two CUBES are congruent if they are identical as strings or if both contain the primitive object $\bar{o}$ (null).

CONGRUENCE- or CONG-objects

(CUBE[u] $\rightarrow$ u | u)

(CUBE[$\langle u \rangle$] $\rightarrow$ $\langle u\bar{o} \rangle$ | $\langle \bar{o} \rangle$, $\langle \bar{o}u \rangle$ | $\langle \bar{o} \rangle$)

(CONG[u | v] $\rightarrow$ v | u)

(CONG[$\langle u \rangle$ | $\langle v \rangle$], CONG[$\langle x \rangle$ | $\langle y \rangle$] $\rightarrow$ $\langle ux \rangle$ | $\langle vy \rangle$)

(CONG[u | v], CONG[v | w] $\rightarrow$ u | w)

Notation: Given objects u and v, we shall generally write $u \approx v$ to denote that $u \mid v$ is a CONG-object.

We now define the negation or complement of the above operation.

NON-CONGRUENCE- or NCONG-objects

$(\rightarrow\langle 0\rangle \mid \langle 1\rangle, \langle 0\rangle \mid \langle X\rangle, \langle 1\rangle \mid \langle X\rangle)$

$(\text{CUBE}[\langle u\rangle] \rightarrow \langle 0u\rangle \mid \langle 0\rangle, \langle 0u\rangle \mid \langle 1\rangle, \langle 1u\rangle \mid \langle X\rangle,$
$\langle 1u\rangle \mid \langle 1\rangle, \langle 1u\rangle \mid \langle 0\rangle, \langle 1u\rangle \mid \langle X\rangle,$
$\langle Xu\rangle \mid \langle X\rangle, \langle Xu\rangle \mid \langle 0\rangle, \langle Xu\rangle \mid \langle 1\rangle)$

$(\text{P}[p], \text{P}[q], \text{NCONG}[\langle u\rangle \mid \langle v\rangle] \rightarrow \langle up\rangle \mid \langle vq\rangle, \langle pu\rangle \mid \langle qv\rangle)$

$(\text{NCONG}[u \mid v] \rightarrow v \mid u)$

$(\text{CONG}[u \mid \langle \bar{o}\rangle], \text{IOX}[p] \rightarrow u \mid \langle p\rangle)$

$(\text{CONG}[u \mid \langle \bar{o}\rangle], \text{NCONG}[u \mid \langle v\rangle], \text{NCONG}[u \mid \langle y\rangle]$
$\rightarrow u \mid \langle vy\rangle)$

Notation: Give objects u and v, we shall generally write $u \not\approx v$ to denote that $u \mid v$ is a NCONG-object.

We turn now to the consideration of LABELLED-CUBES (L-CUBES). We first define an equivalence operation of L-CUBES; to wit, two L-CUBES we considered

to be "equal" if they are identical as B-STRINGS of PL-CUBES up to a reordering of the constituent PL-CUBES.

L-CUBE-EQUALITY-STRONG- or LES-objects

(L-CUBE[u] → u | u)

(L-CUBE[⟨x⟩], L-CUBE[⟨y⟩] → ⟨yx⟩ | ⟨xy⟩)

(LES[x | y], LES[y | z] → x | z)

(LES[x | y] → y | x)

(LES[⟨x⟩ | ⟨y⟩], LES[⟨w⟩ | ⟨u⟩] → ⟨xw⟩ | ⟨yu⟩)

We next define the relation OCCURS (and its complement). The relation we are expressing is that of whether or not a given PL-CUBE occurs in a given L-CUBE.

OCCURS-object

(PL-CUBE[u] → u | ⟨u⟩)

(PL-CUBE[u], L-CUBE[⟨w⟩], OCCURS[u | ⟨w⟩],

L-CUBE[⟨x⟩], L-CUBE[⟨y⟩] → u | ⟨xw⟩, u | ⟨wy⟩, u | ⟨xwy

NOT-OCCURS-object

(LABEL[⟨u⟩], LABEL[⟨v⟩], u ≠ v, IOX[p], IOX [q]

→ ⟨up⟩ | ⟨⟨vq⟩⟩)

(PL-CUBE[u], L-CUBE[⟨v⟩], L-CUBE[⟨w⟩],

NOT-OCCURS[u | ⟨v⟩], NOT-OCCURS[u | ⟨w⟩] → u | ⟨vw⟩)

We can now define the reduction operation.

REDUCTION- or REDUCT-objects

(LABEL[u] → ⟨⟨uX⟩⟩ | X, ⟨⟨ul⟩⟩ | ⟨⟨ul⟩⟩, ⟨⟨u0⟩⟩ | ⟨⟨u0⟩⟩)

(L-CUBE[⟨x⟩], L-CUBE[y], PL-CUBE[u], OCCURS[u | ⟨x⟩],

REDUCT[⟨x⟩ | y] → ⟨xu⟩ | y)

(L-CUBE[⟨x⟩], L-CUBE[y], PL-CUBE[⟨uX⟩],

REDUCT[⟨x⟩ | y] → ⟨x⟨uX⟩⟩ | y)

(L-CUBE[⟨x⟩], PL-CUBE[⟨u0⟩], OCCURS[⟨ul⟩ | ⟨x⟩]

→ ⟨x⟨u0⟩⟩ | ō)

(L-CUBE[⟨x⟩], PL-CUBE[⟨ul⟩], OCCURS[⟨u0⟩ | ⟨x⟩]

→ ⟨x⟨ul⟩⟩ | ō)

(L-CUBE[⟨x⟩], L-CUBE[⟨y⟩], REDUCT[⟨x⟩ | ⟨y⟩], IO[p],

LABEL[u], NOT-OCCUR[⟨up⟩ | ⟨x⟩] → ⟨x⟨up⟩⟩ | ⟨y⟨up⟩⟩)

(L-CUBE[⟨x⟩], REDUCT[⟨x⟩ | ō], L-CUBE[y] → ⟨xy⟩ | ō)

(L-CUBE[⟨x⟩], REDUCT[⟨x⟩ | X], L-CUBE[⟨y⟩],

REDUCT[⟨y⟩ | z] → ⟨xy⟩ | z, ⟨yx⟩ | z)

Note that the above definition also provides an algorithm for computing the REDUCT of any L-CUBE.

We now define

REDUCED-LABELLED-CUBE- or RL-CUBE-object

$$(\text{L-CUBE}[x], \text{REDUCT}[x \mid x] \rightarrow x)$$

Finally we define the operation INTERFACE- or INT-object.

$$(\text{RL-CUBE}[\langle x \rangle], \text{RL-CUBE}[\langle y \rangle], \text{REDUCT}[\langle xy \rangle \mid z] \rightarrow \langle x \rangle \langle y \rangle \mid z)$$

Given RL-CUBES x and y , we shall often write $x \sqcap y$ for their interface; i. e., for that object z such that $\text{INT}[xy \mid z]$. We shall also write $\sqcap[xy \mid z]$ for $\text{INT}[xy \mid z]$.

2. 3 (Partial) Components and circuits

We are now in a position to give initial definitions for the concepts of components and (combinational) cir-

cuits. The definitions we shall give in this subsection will deal with what we shall call partial-components; i. e., components whose behavior may not be specified for all possible combinations of input signals. The notion of a partial-component suffices for the defining of a general concept of a combinational circuit. Certain applications involving don't-care conditions would seem to require the use of the notion of partial-components and partial-circuits; however, in order to provide a straightforward concept of the analysis of a circuit, we shall, in the next two subsections, introduce one definition of a complete-component.

What we wish to do is extract the essential features of Figure 2 (in order to define components) and Figure 3 (in order to define combinational-circuits).

It is convenient to begin by specifying the type of object which corresponds to a row in a table such as that in Figure 2. For this purpose, we shall use LABELLED-SINGULAR-cubes (LS-CUBES). Thus, we use $(\langle a1\rangle\langle b1\rangle \mid \langle c1\rangle)$ to represent the first row

a	b	c
1	1	1

of the table for a two-input AND. However, not all LS-CUBES will or can be used. First of all, we cannot employ LS-CUBES such as $(\langle a1\rangle\langle b1\rangle \mid \langle a1\rangle)$ since the label "a" occurs both as an input and an output label and while this may be a way of representing "feedback," it is clearly out of place in a definition of combinational circuits. Secondly, in order to keep the notation as compact as possible, we will want to restrict ouselves to LS-CUBES $\langle xy\rangle$ where both x and y are REDUCED-LABELLED-CUBES (RL-CUBES). Thus, for the objects corresponding to the rows of the informal representation, we define

ACYCLIC-REDUCED-LABELLED-SINGULAR-CUBE-

or ARLS-CUBE-objects

$$\text{LS-CUBE}[\langle\langle x\rangle\langle y\rangle\rangle], \text{REDUCT}[\langle x\rangle \mid \langle x\rangle], \text{REDUCT}[\langle y\rangle \mid \langle y\rangle],$$
$$\text{REDUCT}[\langle xy\rangle \mid z], \text{LES}[\langle xy\rangle \mid z] \rightarrow \langle\langle x\rangle\langle y\rangle\rangle]$$

Note that what we have done to assure the desired acyclicity (no-feedback) is to make use of the fact that if

x and y have a label in common, then the REDUCT of $\langle xy \rangle$ will be shorter than $\langle xy \rangle$.

We are now in a position to define the concept of a partially specified component. Such a component will first of all be a B-STRING of ARLS-CUBES. However, it is again necessary to introduce additional conditions to insure 1) that there is no feedback, and 2) that the logical function realized by the component is single valued. This second requirement corresponds to the requirement on a table that no two rows specify different output signals for the same input signals. To realize the desired condition, it suffices to specify the correct relationship between pairs of ARLS-CUBES and then employ a lift (from Section 1) to extend it to B-STRINGS of ARLS-CUBES. The desired relation on pairs of ARLS-CUBES is as follows:

PARTIAL-COMPONENT-CONDITION- or

PCCOND-objects

(ARLS-CUBE[x | y], ARLS-CUBE[u | v]

REDUCT[⟨xv⟩ | z], LES[⟨xv⟩ | z]

REDUCT[⟨uy⟩ | w], LES[⟨uy⟩ | w]

$\prod$[⟨x⟩⟨u⟩ | s], $\approx$[s | $\bar{o}$] → (x | y) | (u | v))

(ARLS-CUBE[x | y], ARLS-CUBE[u | v]

REDUCT[⟨xv⟩ | z], LES[⟨xv⟩ | z]

REDUCT[⟨uy⟩ | w], LES[⟨uy⟩ | w]

$\prod$[⟨x⟩⟨u⟩ | s], $\not\approx$[s | $\bar{o}$], $\prod$[⟨y⟩⟨v⟩ | t], $\not\approx$[t | o]

→ (x | y) | (u | v)

Using the above, we then define a partial component as follows:

PARTIAL-COMPONENT- or P-COMP-object

(BSTRING(ARLS-CUBE)[x], □(PCCOND)[x | x] → x)

Informally, a combinational circuit is just a collection of components interconnected in such a manner that there is no feedback. To capture this notion within our formal framework, we define a combinational circuit

to be a B-STRING of P-COMPS such that, to put it somewhat informally, the output labels of a P-COMP in the string only appear as input labels of P-COMPS appearing to its right in the string. To capture this notion in a formal manner, we first define a relation between ARLS-CUBES and then use a combination of lifts to produce the desired definition of combinational-circuit. The actual relation employed tests two things: Given a pair of ARLS-CUBES, it tests to see 1) that the second does not "feed back" to the first, and 2) that they have distinct output labels (this is to insure that "physically distinct" circuits have "physically distinct" outputs). The relation is formally written as follows:

COMBINATIONAL-CIRCUIT-CONDITION- or

CC-COND-object

(ARLS-CUBE[x|y], ARLS[CUBE[u|v],

REDUCT[⟨yv⟩|z], LES[⟨yv⟩|z],

REDUCT[⟨xv⟩|w], LES[⟨xv⟩|w] → (x|y)|(u|v)) .

We then define

PARTIAL-COMBINATIONAL-CIRCUIT- or

PC-CKT-object

(B-STRING(P-COMP)[x], Δ(□(CC-COND))[x])

Note that the □-lift extends CC-COND to a relation between P-COMPS and the Δ-lift extends the new relation to a predicate on B-STRING(P-COMP).

2.4 Completely specified components and circuits

In the preceding section we defined the class of objects corresponding to partially specified components and circuits. In this section we will give one definition for completely specified components and circuits (those for which output signals are specified for every possible combination of input signals). This definition will be employed in the next section to define (give an algorithm for) analyzing such completely specified circuits.

The definition of completely specified circuit given

in this section will be somewhat stronger than necessary. That is, the definition will consist of a test which will recognize as completely specified only those PARTIAL-COMPONENTS which are of a particular form. We anticipate that a more general definition will be desirable in later papers dealing with the synthesis of circuits; however, the definition given here is sufficient for the analysis of circuits. Informally speaking, what we shall require of a PARTIAL-COMBINATIONAL-CIRCUIT is that each of its PARTIAL-COMPONENTS correspond to a table which covers every possible combination of input signals and that in each "row" of the table the value of each output be specified (i.e., no X's are to occur on the right side of the table).

Let us start by developing the part of the test which determines if every possible combination of input signals is covered in the table corresponding to a PARTIAL-COMPONENT. Informally, this means we want to check to see if the left side of the table contains every possible input combination under the interpretation given in the introduction of this paper. Consider the left side of the

table corresponding to a PARTIAL-COMPONENT. Under the rule (given in the introduction), the X's in this part of the table can be replaced by both 1's and 0's and if all possible such replacements were made, the new table (with no X's) would be the complete listing of input signal combinations for which the operation of the circuit is specified. Clearly, if there are many inputs, such an expansion of the table is impractical (for 20 inputs such a listing would contain about one million entries). To avoid such an expansion, we employ the #-product (sharp-product) developed by Roth [R-1]. First, we represent the left side of the table corresponding to a PARTIAL-COMPONENT by an

RL-COVER

(B-STRING(RL-CUBE)[u] → u) .

We then define the #-product on such RL-COVERS. As can be seen (by reference to [R-1]), such an RL-COVER u covers every possible input combination only if $X \# u = \bar{0}$. We now turn to the job of defining the

#-product within our formal system.

Since the #-product will introduce $\bar{o}$ values, we will need the following trivial generalization of RL-COVERS.

RL-NULL-COVER- or RLN-COVER-objects

$(\rightarrow \langle \bar{o} \rangle)$

$(\text{RL-COVER}[x] \rightarrow x)$

$(\text{RLN-COVER}[\langle x \rangle], \text{RLN-COVER}[\langle y \rangle] \rightarrow \langle xy \rangle)$

Correspondingly, we will need the following operation to delete $\bar{o}$'s from RLN-cubes.

NULL-DELETE- or ND-object

$(\rightarrow \langle \bar{o} \rangle \mid \langle \bar{o} \rangle)$

$(\text{RL-COVER}[x] \rightarrow x \mid x)$

$(\text{RL-COVER}[\langle x \rangle] \rightarrow \langle x\bar{o} \rangle \mid \langle x \rangle)$

$(\text{RLN-COVER}[\langle x \rangle], \text{RLN-COVER}[\langle y \rangle]$

$\text{ND}[\langle x \rangle \mid \langle \bar{o} \rangle], \text{ND}[\langle y \rangle \mid \langle v \rangle] \rightarrow \langle xy \rangle \mid \langle v \rangle, \langle yx \rangle \mid \langle v \rangle)$

$(\text{RLN-COVER}[\langle x \rangle], \text{RLN-COVER}[\langle y \rangle]$

$\text{ND}[\langle x \rangle \mid \langle u \rangle], \text{ND}[\langle y \rangle \mid \langle v \rangle]$

$\text{RL-COVER}[\langle u \rangle], \text{RL-COVER}[\langle v \rangle] \rightarrow \langle xy \rangle \mid \langle uv \rangle)$

To define the #-product it is convenient to introduce the following operation which permits one to append a PL-CUBE to every RL-CUBE in an RL-COVER.

APPEND

(RL-CUBE[⟨x⟩], IO[p], LABEL[y],

NOT-OCCUR[⟨yp⟩ | ⟨x⟩], → (⟨yp⟩ | ⟨⟨x⟩⟩) | ⟨⟨x⟨yp⟩⟩⟩)

(PL-CUBE[y], RL-COVER[⟨x⟩], RL-COVER[⟨u⟩]

APPEND[(y | ⟨x⟩) | ⟨z⟩], APPEND[(y | ⟨u⟩) | ⟨w⟩]

→ (y | ⟨xu⟩) | ⟨zw⟩)

The #-product between individual RL-CUBES (and $\bar{o}$'s) is then defined as follows:

SHARP-OF-CUBES- or SHRPC-objects

(RL-CUBE[⟨x⟩], RL-CUBE[⟨y⟩], REDUCT[⟨xy⟩ | $\bar{o}$]

→ (⟨x⟩⟨y⟩ | ⟨x⟩)

(RL-CUBE[u] → $\bar{o}$u | $\bar{o}$, u$\bar{o}$ | u)

(LABEL[y] → ⟨X⟨y1⟩⟩ | ⟨⟨y0⟩⟩, X⟨⟨y0⟩⟩ | ⟨⟨y1⟩⟩)

($\bigsqcap$ [xy | z], LES[x | z] → xy | $\bar{o}$)

(SHRPC[⟨x⟩⟨y⟩ | z], LABEL[u], IO[w],

RL-CUBE[⟨x⟨uw⟩⟩], RL-CUBE[⟨y⟨uw⟩⟩],

APPEND[⟨uw⟩⟨z⟩ | ⟨v⟩] → (⟨x⟨uw⟩⟩⟨y⟨uw⟩⟩ | v))

(SHRPC[⟨x⟩⟨y⟩ | z], RL-COVER[⟨z⟩], LABEL[u],

NOT-OCCUR[⟨u0⟩ | ⟨xy⟩], NOT-OCCUR[⟨ul⟩ | ⟨xy⟩]

→ (⟨x⟩⟨y⟨ul⟩⟩ | z⟨x⟨u0⟩⟩), (⟨x⟩⟨y⟨u0⟩⟩ | z⟨x⟨ul⟩⟩))

(SHRPC[⟨x⟩⟨y⟩ | $\bar{o}$], LABEL[u],

NOT-OCCUR[⟨u0⟩ | ⟨xy⟩], NOT-OCCUR[⟨ul⟩ | ⟨xy⟩]

→ (⟨x⟩⟨y⟨u0⟩⟩ | ⟨x⟨ul⟩⟩), (⟨x⟨⟩y⟨ul⟩⟩ | ⟨x⟨u0⟩⟩))

The above #-product can now be lifted to an operation on RL-COVERS as follows:

SHARP-OF-COVERS- or SHCOV-objects

(IA(FΣ2(IA(SHRPC)))[⟨y⟩x | ⟨z⟩], ND[⟨z⟩ | ⟨w⟩] → ⟨y⟩⟨x⟩ | ⟨w⟩)

(SHCOV[⟨x⟩⟨y⟩ | ⟨z⟩], RLN-CUBE[u], SHCOV[⟨z⟩⟨u⟩ | ⟨w⟩]

→ ⟨x⟩⟨yu⟩ | ⟨w⟩)

Note: The definition schema IA is employed here in order to arrange the variables in the desired order to employ FΣ2

and then to rearrange them so as to have SHCOV[xy | z] correspond to x # y = z .

The above definitions provide the machinery necessary for the first part of the test. We turn now to setting up the machinery for the second part of the test--for checking that there "are no X's on the right side of the table." Our procedure here is to check that the LABELS are the same on the "right side" of every ARLS-CUBE in the PARTIAL-COMPONENT. To do this, we define the following types of objects:

LABEL-EXTRACT- or LE-object

(PL-CUBE[⟨⟨x⟩y⟩] → ⟨x⟩)

The above will extract the label from a PL-CUBE. We can lift it to extract the LABELS from the right side of an RL-CUBE as follows:

OUTPUTS-RL-CUBE- or OAC-object

$$(\text{RL-CUBE}[x], F\Sigma 1(\text{LE})[x \mid u] \rightarrow x \mid u)$$

Next, we can define equality of B-STRINGS of LABELS.

EQUALITY-LABEL-STRING- or ELS-object

$$(\text{B-STRING(LABEL)}[u] \rightarrow u \mid u)$$

$$(\text{B-STRING(LABEL)}[\langle u \rangle], \text{B-STRING(LABEL)}[\langle v \rangle] \rightarrow \langle uv \rangle \mid \langle vu \rangle)$$

$$(\text{ELS}[u \mid v], \text{ELS}[v \mid w] \rightarrow u \mid w, v \mid u)$$

Finally, defining the function

ID-object

$$(\text{OBJECT}[u] \rightarrow u \mid u)$$

We can now put these definitions together and define:

COMPLETELY-SPECIFIED-COMPONENT- or CS-COMP

$$(\text{P-COMP}[u], F\Sigma 1(\text{FP2(LE)})[u \mid v], \Box(\text{ELS}[x \mid x], F\Sigma 1(\text{FP1(ID)})[u \mid w], \text{SHCOV}[\langle X \rangle w \mid \langle \bar{o} \rangle] \rightarrow u)$$

Then we also get:

COMPLETELY-SPECIFIED-CIRCUIT- or CS-CKT

(B-STRING(CS-COMP)[x], Δ(□(CC-COND))[x] → x)

directly from the definition of PC-CKT.

2.5 The analysis of circuits

Given all the apparatus now at our command, it is very easy to present an algorithm for analyzing CS-CKT's. We first define PAIR-DELETE- or PDEL-objects

(PAIR[⟨⟨x⟩⟨y⟩⟩] → ⟨⟨x⟩⟨y⟩⟩ | ⟨xy⟩)

and from this,

CIRCUIT-SKELETON-object

(PS-CKT[x], FΣ1(FΣ1(PDEL))[x | y] → x | y) .

Note that if x is a PS-COMP and CIRCUIT-SKELETON[⟨x⟩ | y], then y is (corresponds to) the analysis table of the circuit

consisting of x alone.

Now we define:

ANALYSIS

(CS-CKT[x], CIRCUIT-SKELETON[x|y],

FII2(F□2(⌐)[y|z], ND[y|w] → x|w) .

3. CONCLUDING REMARKS

The definitions and algorithms given in the preceeding section serve to illustrate that we can employ the α-object calculus to define the basic entities and operations of switching theory. It should be clear that the α-object calculus, as a notation, provides a precise way to write down the definitions and algorithms that we need. What we have not shown in this paper is that this approach provides anything beyond precision and a certain mathematical economy of initial means. In particular, we have not shown that the α-object calculus can be gainfully employed to facilitate proof of the correctness of definitions, or the validity of algorithms.

It is our contention that the α-object calculus can be gainfully employed to develop the theory (i. e., theorems and proof) as well as the definitions and algorithms of switching theory. However, we believe that the most fruitful approach to this problem is through a study of the underlying structure of the α-object calculus. Such a study should lead to precise notions of "data structure," "definition," :algorithm," "application of algorithms,"

and should also lead to an associated proof theory. This would provide a general theory of algorithms and data-structures of interest in itself and with many applications including, of course, the theory of switching as begun in this report. In particular, the methods for dealing with α-objects (and formal definitions and definition strings) in proofs should provide a uniform and precise approach for proving the theorems of switching theory.

Preliminary research has led a natural generalization of the calculus of α-objects to similar calculi over algebras with finitely many operators and defining relations. Viewed this way, the α-object calculus of this paper is defined over a calculus with one binary operation (concatenation), one 1-ary operation (angle-bracketing) and four 0-ary operations (the constants 0, 1, X, and $\bar{o}$); and with one defining relation (concatenation is associative). The more general approach allows one to deal with problems arising from changes of notation, the relative power of diiferent notations, and with the general notions of mathematics, such as function and relation. It also appears to facilitate the application

of the results of recursive function theory to the calculi and to the problems (decidable or undecidable) concerned with the optimization and classification of algorithms. This is relevant to the problems of proving the validity of algorithms as the form (or classification) of an algorithm (or formal definition) is closely connected with what can be proved, or how something can be proved, about that algorithm. Because of the underlying finiteness of switching theory (as reflected by the fact that there "always" exist exhaustive algorithms for finding solutions), it is conceivable that switching theory can be formulated in some manner which particularly facilitates proofs (and avoids most, if not all, questions of undecidability). However, the proof of the existence, and the finding of such a formulation, rests on further investigation of the underlying calculi.

The α-object calculus, and the more general calculi, also provide a means for providing a rigorous formulation of the F-notation [R-W-2]. In particular, these calculi can be employed to give rigorous semantics to any particular F-notation. By combining the α-object

calculus and the F-notation, one should be able to produce a rigorous, convenient, and uniform language in which to describe all the switching algorithms given in the references of this paper.

REFERENCES

[R-1] Roth, J. P., "Algebraic topological methods for the synthesis of switching systems, I," Transactions of the American Mathematical Society, Vol. 88, No. 2 (July 1958) 301-326.

[R-W-1] Roth, J. P., and E. G. Wagner, "Algebraic topological methods for the synthesis of switching systems, Part III, Minimization of nonsingular Boolean trees," IBM Journal of Research and Development, Vol. 4, No. 4 (October 1959) 326-344.

[R-K] Roth, J. P., and R. M. Karp, "Minimization over Boolean graphs," IBM Journal of Research and Development, Vol. 6, No. 2 (April 1962) 227-238.

[R-2] Roth, J. P., "Diagnosis of automata failures: A calculus and a method," IBM Journal of Research and Development, Vol. 10, No. 4 (July 1966).

[R-W-2] Roth, J. P., and E. G. Wagner, "A calculus and an algorithm for a logic minimization problem together with an algorithmic notation," Chapter II of this report.

[S] Smullyan, R. M., "Theory of formal systems," Annals of Mathematics Studies Number 47, Princeton University Press, Princeton, New Jersey (1961).

An APL Program for the Multiple Output 2-Level Minimization Problem

by

Leon S. Levy

Research Division
Yorktown Heights, New York

TABLE OF CONTENTS

1. INTRODUCTION

The APL/360 programs for the multiple-output two-level minimization algorithm are an initial version which serves two purposes:

1) Use of the programs should ease the learning of the algorithm since many examples are readily available; a (trivial) program has been written to generate test examples;
2) Having an operating program available, more efficient versions may be prepared and test results validated.

The programs follow very closely the F-notation formulation of the algorithm given in Reference 1. A brief description of the algorithm follows:

The solution is built up recursively. Initially, the prime cubes are computed, and then an 'extremal' program EBAR is called. If there are any extremals, the 'distinguished' part of the extremals is added to the solution and removed from the prime cube list. Then the extremal program is reentered to find the next order extremals (this is analogous to an onion-peeling process). When no extremals are found, an arbitrary choice is made in the branching program, BBAR, which then calls EBAR to build up each of the two solutions. Multiple branching can lead to a large solution tree, where EBAR and BBAR are repeatedly called recursively.

---

[1] A Calculus and an Algorithm for a Logic Minimization Together with an Algorithm Notation - J. Paul Roth, E. G. Wagner.

The program contains, as modules, the identifiable sub-algorithms such as the "sharp-algorithm" for computation of the prime cubes, the extremal computation, and the cost evaluations both in the less-than operation and in the branch selection. Portions of the program may thus be changed to be more efficient in speed or storage without revising the whole program, and statistics on the relative performances easily obtained. Also, the sub-algorithms are useable independent of the overall algorithm.

To facilitate the modular usage and revision of the algorithm a brief programmer's manual is included in Section 4. The syntax and semantics of each function in the program is described.

Section 2 describes the structure of the program. Section 3 provides an introduction to its use.

2. PROGRAM STRUCTURE

The program takes as input two singular covers, one covering the care complex of the problem, and the other covering the don't care complex, and proceeds to compute recursively the minimum cost solution. The output is a singular cover representing the minimum solution.

A singular cube is represented in the program in one of three forms:

a) As an alphameric vector of the form $'a_1 \ldots a_m | b_1 \ldots b_n'$

$$a_i \in 0, 1, x, \phi$$

$$b_i \in 1, x$$

m = the number of input coordinates, n = the number of output coordinates.

b) As a 1 by (m + n + 1) matrix q of the same components (since the APL system distinguishes between an n-element vector and a 1 x n element matrix).

c) As a null vector, in which case it identifies the empty cube.

The "universal cube" has a_i = x(all i) and b_j = 1 (all j). A cube in which b_j = x for all j is the empty cube. A singular cover is a k x (m + n + 1) alphameric matrix, each of whose rows represents a singular cube.

Examination of the F-notation formulation of the algorithm shows that most functions can be written as functions of four variables each of which is a singular cover.

C - a cover of the care complex

D - a cover of the don't care complex

S - the singular cover of the solution

Z - the set of prime cubes.

The APL language allows functions of 0, 1, or 2 variables so that it is necessary to group the variables, so that a function of four variables can be written as an apparent function of only two variables. Because of this, and also because of the way the APL interpreter treated local variables, the following single variable array ('GΔ' array) was developed which contains within it the C, D, S, Z variables. The GΔ array is a ρ by (m + n + 1) matrix in which four 'tag' rows; 'GΔC' 'GΔD', 'GΔS' and 'GΔZ' are used as markers to separate the variables, and where ρ is 4 plus the number of singular cubes in C, D, S, Z . The starting GΔ array is

formed by an initialization function which accepts as arguments the variables C, D of the problem.

The three main functions in the program are MXBAR, EBAR, and BBAR in direct correspondence with the F-notation formulation of the minimization algorithm. MXBAR accepts as an argument the initial value of the GΔ array, and, if S covers C, returns S as the result. Otherwise, if S does not cover C, it computes the prime cubes using the SHARPALG function, appends the set of prime cubes to the GΔ array, and calls EBAR with the updated GΔ array as an argument. Note that if part of the solution is known, the initial GΔ array can be composed to include it (although the INITIAL function does not provide this feature), in which case the solution might converge more rapidly. However, if such an initially introduced solution contains terms which are not part of the minimum solution, they would never be subsequently removed (either by the program, or, correspondingly, by the F-algorithm.)

EBAR and BBAR are syntactically similar to MXBAR: their input argument is a GΔ array and their output is a singular cover. EBAR accepts the current GΔ array as argument, and, if the "solution" part of the GΔ array covers the "care" part of the GΔ array, returns the solution part as its value. (The OF function extracts a specified part of the GΔ array as follows: Let A be the GΔ array then GΔC OF A will return the care part of the array. GΔC is the name of a tag vector which specified by the INITIAL function.)

If the current solution is incomplete then the less than operation is performed, using the XTX and XUX functions, (The less than operation on A is A ←(XTX A) X U X A) and the extremals of the remaining prime cubes are computed using the EXT function. If the set of extremals computed is non-empty, then EBAR is entered recursively with modified argument. The modification performed by the DELTA function adds the distinguished vertices of the extremals to the solution and removes them from the current extremals.

If the set of extremals computed in EBAR is empty, then the branching process is initiated using the BBAR function with the current GΔ array as argument. On entering the BBAR function a selection of an output vertex is made by the user, and the solutions with and without that output vertex are explored, and the lesser cost solution is chosen. The development of the alternative solutions is made by generating updated GΔ arrays for the alternative choices and executing the EBAR function for each. Of course, the EBAR function along either path may branch again and BBAR can thus be called recursively.

Ultimately each path of the recursion tree must terminate, since the algorithm always yields a solution.

3. INTRODUCTION TO USE

In this section, we assume that the reader has an APL terminal available, and knows how to use the APL system. Making use of the descriptions in Section 4, it is suggested that the functions be tried out in the following order to gain some familiarity with the program elements: XBX, XDX, XEX, INTF, XIX, XJX, XKX, XMX, XSX, SHARP, SHARPALG, CCAT, IN, MINUS, RESID. (The set of examples given in Section 4 may also be used to verify proper operations.) The functions in the program make use of a set of global variables, which are initialized by the INITIAL function. Referring to the description in Section 4, it may be seen that

$$(\iota 0)\text{INITIAL 'XXX}\underset{m}{\ldots}\text{X}\,|\,\text{XX}\underset{n}{\ldots}\text{X'}$$

will set up the global variables for singular cubes with m input coordinates and n output coordinates. Wherever the dimensions of the singular cubes are to be changed, a new INITIAL function should be executed.

If a random problem is desired, the function G←AGΔDTESTB should be executed which will specify C as the initial GΔ array, A, B, being specified according to their syntax as described in Section 4. If a known problem is to be run, the INITIAL function may be executed directly. As stated, this function has C, D as arguments. If an initial value of S is to be specified as well, then the CCAT function is used to add it to the GΔ array.

The program is then executed using MXBAR with the generated GΔ array as argument. (Note that providing the GΔ array directly, without use of the INITIAL function will cause an error condition since certain global variables are set up as side effects of the INITIAL function.)

In executing MXBAR, the program will trace EBAR by printing EBAR followed by its argument whenever EBAR is entered, and similarly BBAR will be traced. This trace is helpful in visualizing the recursive structure of the execution. The APL printout during execution is shown in the set of examples in Section 4.

During the BBAR function execution, the program halts and waits for the manual input of a two element numeric vector. The first number specifies which element of Z of the GΔ array is to be selected and the second number selects the vertex of the output part. Suppose Z is given by:

1 x 0 | 1 x
1 1 x | 1 1
x 1 1 | 1 1

Then an input of 2 2 will select 1 1 x | x 1 as the singular cube to be added to the solution along the S^g path of the branching function. To facilitate usage, all of the functions described in Section 4 have been combined as a function group in APL, designated MOALG. The functions actually required in the algorithm itself have also been grouped as MOALG1, which does not include initialization, test case generation.

The less than operation was programmed according to an early version of the algorithm and does not correspond to the final formulation. The programmed version is $u < v$ if cost $(S \cup \{U\}) \geq$ cost $(S \cup \{v\})$ and $(u \# (S) \# v = \overline{0}$. The proper formulation, given in [1] is:

If for every coface u' of u (including the case $u' = u$) there exists a coface v' of v such that $(u' \# (D \cup S)) \# v' = \overline{0}$ and cost $(S \cup \{v'\}) \leq$ cost $(S \cup \{u'\})$. The cost is evaluated in a function called COST, which may be revised or rewritten to conform to different technological factors. The programmed version is described in Section 4 and examples are given at the end of that section; it envisions a two-level gating structure and adds a unit cost for each input variable occurrence, and a weighted sum of the first level outputs depending on their fanout to the second level gates.

4. PROGRAMMER'S MANUAL

In this section, the APL functions in the program are listed alphabetically. Following each function, the syntax of the function call and the result is described. Then a description of the function is given.

Note that the syntax as described is restricted to the intended use, and is not the broadest possible syntax for the given functions. When a function is used with a broader syntax, the semantics of the function is not necessarily as described here. As a trivial example, if the function INTF is executed, all that is required syntactically by the program is that the argument be commensurate cubes of length $\geq \underline{S} \lceil \underline{T}$;

moreover, B is a global vector which can be extended, by concatentation to allow INTF to accept syntactically as its first argument any commensurate alphameric vector satisfying the length restriction noted above, without altering the syntax or intended semantics of INTF.

A set of examples is given at the end of this section.

BBAR

Syntax: C ← BBAR A

A is a GΔ array

C is a singular cover

Semantics: BBAR is the branching algorithm, B(C, D, S, Z). When BBAR is entered 'BBAR' followed by the GΔ array is printed out and input is requested from the user. The input is a two component numerical vector, whose first component denotes which singular cube of Z is to be chosen, and whose second components selects the distinguished output of this singular cube. Two GΔ arrays are now formed corresponding to the two paths of the branch and EBAR is computed for each GΔ array. The costs of the computed singular covers along the paths are compared and the solution of lesser cost is chosen.

Explication: Local variable G corresponds to g, H to h, and Q to f, N to S^g and 0 to $S^{\underline{g}}$.

CCAT

Syntax: C ← A CCAT B

A is any matrix, or vector;

B is any matrix, or vector;

Either A or B is empty (⍳0) or

A and B have the same number of columns.

C is A if B is empty

C is B if A is empty

C is a matrix

Semantics: If A, B are matrices having the same number of columns, A CCAT B is a matrix of the form $\begin{bmatrix} B \\ \vdots \\ A \end{bmatrix}$. If A or B is a vector it is considered as a matrix of one row.

COST

Syntax: C ← A

A is a singular cover

C is a numeric scalar

Semantics: COST computes the cost of the solution part S of A. First, interface any cubes of S that have the same input (XAX function). Then the cost of each cube is computed as the sum of its nonvacuous inputs added to a function of its output components. The implemented function charges a unit cost for up to 3 output components, and two units of cost for 4-9 output components. Finally, the costs of all the cubes are summed.

DELTA

Syntax: C ← E DELTA A

A is a GΔ array

E is a singular cover

C is a GΔ array

Semantics: A is the current GΔ array, and E is the current set of extremals. The local variable G, corresponding to f in the F-algorithm notation, covers the distinguished vertices of the extremals to be added to the current solution. I (cf. program listing, Section 5) covers the other vertices of the extremals which are not added to the solution. C is the updated GΔ array where the following operations have been performed: C replaced by C#F, D replaced by DUSUF, S replaced by SUF, and Z replaced by Z-EU(E#αF).

EBAR

Syntax: C ← EBAR A

A is a GΔ array

C is a singular cover

Semantics: If S, the solution part of A, covers the vertices of C, the care complex of A, then S is returned. Otherwise, the less than operation on A is performed by A ←(XTX A) SUX A. Then, if the resulting A has no extremals, BBAR is called to initiate the branching process. Otherwise, the DELTA operation is performed to add to S the distinguished vertices of the extremals, and EBAR is applied to the resulting GΔ array.

EXT

Syntax: C ← EXT A

A is a GΔ array

C is a singular cover

Semantics: EXT computes the extremals by the formula

$EXT = \{ z \in Z \mid z \# (S \cup D \cup (Z - z) \neq 0 \}$,

by executing XAY with Z and S U D as arguments.

GΔCTEST

Syntax: C ← A GΔCTEST B

A is a two component numeric vector

B is a number

C is a cover

Semantics: GΔCTEST is a random test generator which generates a singular cover of B singular cubes (after subsuming) having m inputs and n outputs each where m is the first component of A and n is the second component of A.

GΔDTEST

Syntax: C ←A GΔDTEST B

A is a two component vector

B is a number

C is a GΔ-array

Semantics: GΔDTEST generates a test case, utilize GΔCTEST, initializes the GΔ array and executes the multiple output minimization algorithm. A [1] is the number of inputs. A [2] is the number of outputs. B is the number of cubes in the cover of the care complex.

IN

Syntax: C ←A IN B

A is a singular cube

B is a singular cover

C is a number

Semantics: If A is in B, C is the row number of A in B. If A is not in B, C is m+1, where m is the number of rows in B.

INITIAL

Syntax: C ←A INITIAL B

A, B are singular covers

C is a GΔ-array

Semantics: A is a singular cover for the complex of don't cares, which may be empty. B is a cover for the complex of cares, which must be non-empty. INITIAL forms the starting GΔ array and sets up the global vectors GΔC, GΔD, GΔS, GΔZ which are used as tags, the global scalars D, L, M, S, T, U, and the global vectors A, B, E, F, G, H. (If there are no don't cares, ⍳0 is entered for A.)

INTF

Syntax: C ←A INTF B

A, B, C are singular cubes

Semantics: C is the interface of A and B.

MINUS

Syntax: C ←A MINUS B

A, B, C are singular covers.

Semantics: C is A with the cubes of B deleted.

MXBAR

Syntax: C ←MXBAR A

A is a GΔ-array

C is a singular cover.

Semantics: If S, the current solution, covers the vertices of the care complex, then the result is given by S. Otherwise, the set of prime cubes is computed as #alg(CUD) by the SHARPALG function, and appended to the GΔ array. EBAR is now applied to the resulting GΔ array.

OF

Syntax: C ← AOF B

A is a label in GΔ

B is a GΔ-array

C is a cover

Semantics: C is the cover of singular cubes following the label A up to, but not including, the next label.

PF

Syntax: C ←A PF B

A is the input part of a singular cube

B is a GΔ array

C is 0 or 1

Semantics: C is a predicate, which is 1 when A is the input part of some cube in the solution part of B; otherwise C is 0.

RESID

Syntax: C ←RESID A

A is a matrix

C is a matrix or an empty vector

Semantics: C is the result of deleting the top row of A. If A is a vector or a matrix of one row, then C is empty.

SHARP

Syntax: C ←A SHARP B

A, B, C are singular covers

Semantics: SHARP is a recursive function which forms A # B, as follows: if A or B is empty, then C is given by A. If B is a single singular cube, then A # B is computed by the XSX function. If B is a cover consisting of more than one singular cube, $B = \{B_1, B_2, \ldots, B_n\}$,

$$A \# B = A \# \{B_1, B_2, \ldots, B_n\}$$
$$= (A \# B_1) \# \{B_2, \ldots, B_n\}.$$

SHARPALG

Syntax: C ← SHARPALG A

C, A are singular covers

Semantics: SHARPALG computes the prime cubes of A where A is a cover of some singular cubical complex. First, a universal singular cube B is formed, all of whose input components are 'x', and all of whose output components are '1'. Then C is given by $B \# (B \# A)$.

XAX

Syntax: C ← XAX A

A is a singular cover

C is a singular cover

Semantics: C is obtained from A by interfacing any singular cubes which have the same input part, adding their interface and deleting these particular cubes.

XAY

Syntax: C ← A XAY B

A, B, C are singular covers

Semantics: XAY is used in the extremal computation and forms the singular cover $\{a \in A \mid a \# (B \cup (A - a) \neq \emptyset\}$.

XBX

Syntax: C ← A XBX B

A, B are commensurate cubes of elements '0', '1', 'x'

C is a cube of elements '0', '1', 'x', 'ϕ'

Semantics: C is the interface of two non-void cubes, where corresponding inputs are interfaced according to the following rule:

	0	1	x
0	0	ϕ	0
1	ϕ	1	1
x	0	1	x

XDX

Syntax: C ← A XDX B

A, B are commensurate cubes of '0', '1', 'x'

C is a cube of '0', '1', 'x', '0'

Semantics: C is a cube whose elements are formed in accordance with the following rule for element composition:

a \ b	0	1	x
0	x	ϕ	0
1	ϕ	x	1
x	x	x	x

XDX is used to form the output part of the SHARP product of singular cubes.

XEX

Syntax: C ← A XEX B

A, B cubes

C is (numeric) 0 or 1

Semantics: C is a predicate which is 1 if B is a face of A, or B is equal to A, 0 otherwise.

XIX

Syntax: C ←A XIX B

A, B are cubes of '0', '1', 'x'

C is a cover of cubes

Semantics: If A#B ≠ 0, then C is A#B

If A#B = $\overline{0}$, then C is empty.

XJX

Syntax: C ← A XJX B

A, B are singular cubes

C is a cover (possibly empty)

Semantics: C is A#B.

Let $A = a_1 | a_2$
$B = b_1 | b_2$
$E = a_2 \star b_2$

where { A is a singular cube; a_1 is the input part; a_2 is the output part

☆	1	x
1	x	1
x	x	x

defines ☆

1. $a_1 \sqcap b_1 = \overline{0}$, then C is equal to A. (Line [9] branches to line [19].

2. $(a_1 \sqsubset b_1)$ or $(a_1 = b_1)$, then
   i) E is all x's C is empty
   ii) E is not all x's C is $a_1 | E$.

(Line [10] branches to line [21] which terminates in case (i) or continues to line [22] in case (ii).

3. $b_1 \sqsubset a_1$, then
   i) C includes $a_1 | E$, if E is not all x's, and
   ii) C includes a term for each input that has an x in A and a 0, or 1 in B.

(Line [12] forms (1), or line [11] branches around it, as appropriate. Lines [13-16] form part (ii). Line 17 restructures C.)

XKX

Syntax: C ← A XKX B

A, B are singular cubes

C is (numeric) 0 or 1

Semantics: C is a predicate which is 1 iff B properly contains A or B = A.

Let A = a|b

B = c|d

Then B properly contains A iff

(c $\sqsubset$ a and (b $\sqsupset$ d or b = d))

or

(d = a and b $\sqsupset$ d).

If A XKX B = 1, then in the subsuming function XMX, A may be subsumed by B.

XMX

Syntax: C ← XMX A

A is a singular cover

C is a singular cover

Semantics: XMX performs the subsuming operation on A.

Explication: Each singular cube in A is compared with each other cube by a "double DO-loop". J indexes the outer loop, I indexes the inner loop. As the solution is being built up, the J^{th} cube of A is compared with successive cubes of C, (line [6]). If A [J;] is contained in C [I;], then I is reset and J is increased (line [9]. If not, then if C [I;] is replaced by A[J;] (line [7], then C [I;] is replaced by A[J;], line [10] and I is reset and J is increased.

If neither C [I;] contains A [J;], or vice versa, I is increased until C is exhausted, at which time A[J;] is added to C, J is increased and I reset.

XNX

Syntax: C ← XNX A

A is a vector or matrix

C is a matrix

Semantics: If A is a matrix then C is given by A. If A is a vector, then C is a restructured into a single row matrix.

XQX

Syntax: C ← XQX A

A is a vector or matrix

C is (numeric) 0 or 1

Semantics: C is a predicate which is 1 if x is a vector or a matrix with one row.

XSX

Syntax: C ← A XSX B

A is a non-empty singular cover

B is a non-empty singular cube

C is a singular cover

Semantics: C is A#B

Explication: In this recursive computation of $A \# B$, where B is a singular cube, and A is a singular cover, A is checked first to see if it contains exactly one cube, (line [2]), in which case the sharp product is computed by XJX (line [7]). Otherwise, letting $A = \{a_1, a_2, \ldots, a_m\} = \{a_1, A_1\}$ where $a_1, \ldots, a_m$ are the singular cubes of A and A_1 is A with a_1 removed; $A \# B = (a_1 \# B) \cup (A_1 \# B)$ is given by lines 3-6.

XTX

Syntax: $C \leftarrow$ XTX A

A is a GΔ-array

C is a (numeric) matrix

Semantics: C is a nx1 matrix where n is the number of cubes in Z. $C[i;1]$ is the cost of Z_i, the i^{th} singular cube of Z. The cost function used is as follows:

Let $Z_i = a|b$

If a is the input part of some singular cube in S then the cost of a is 0. Otherwise, the cost of a is the number of nonvacuous inputs. The cost of b is a function of the number of output coordinates k that have the assignment 1, namely:

cost $b = 1$ $k = 1, 2, 3$

$= 2$ $k = 4, 5, 6, 7, 8, 9$

Explication: The cost function of the number of outputs is given by B in line [3].

XUX

Syntax: C ← Z XUX B

A is a (numeric) (m x 1) matrix where m is the number of singular cubes in the Z component of B. B is a GΔ array, whose Z-component has at least two singular cubes. C is a GΔ array.

Semantics: The first argument is XTX B which specifies the incremental cost for each singular cube of Z.

XUX

Semantics: A is a singular cover, consisting of two singular cubes, $A = \{a_1, a_2\}$, and B is a singular cover. In the "less than" operation, B is the "don't care" singular cover. C is a predicate which 1 iff $(a_1 \# B) \# a_2$ is empty; i.e. that a_2 covers all vertices of a_1 which are not in the don't care complex.

XVX

Syntax: C ← A XVX B

A is a set of two singular cubes

B is a set of singular cubes

C is a (numeric) 0 or 1

Semantics: Let $A = \{a_1, a_2\}$. C is a predicate which is 1 if $(a_1 \# B) \# a_2 = \emptyset$ (XVX is used in the 'less than' computation).

XWX

Syntax: C ← XWX A

A is a non-empty singular cover

C is a cube of elements '1', 'x'

Semantics: C is the interface of the output parts of all the singular cubes of A.

Examples

The following set of examples shows the operation of the various functions within the program as well as the overall program operation. The examples are intended to be self-explanatory and all variables used are defined.

5. PROGRAM LISTING

The APL printout of the function definitions for all of the programs in the M. O. Algorithm are listed below.

Example 1. This example is taken from Appendix 1 of Reference [1]. The initial problem so consists of six care conditions, listed under GΔ C, no don-t-care conditions, so that a blank follows GΔ D, no initial solution GΔ S and no prime cubes GΔ Z initially specified. A minimum cover S_1 for this problem is found invoking the function MXBAR, by the statement

S1 ← MXBAR S0

As described above and in [1] EBAR is executed, first computing the preim cubes Z to form the new GΔ Z array (GΔ C, GΔ D, and GΔ S remain the same.

Extremals are computed, whose results are exhibited prior to the next execution of EBAR: here the new GΔ array shows a new GΔ C, GΔ D, GΔ S and GΔ Z. In this second execution of EBAR no new extremals are formed, so that the branching function BBAR is called: Below BBAR is listed the same GΔ array as shown at the second execution of EBAR. The program allows the user a choice of singular cube and its distinguished face from the GΔ C array: here the choice ⍳1 is made, to choose the second cube and first coordinate.

EBAR is then executed as a function called in BBAR and a new set of extremals computed. No further choices are made: EBAR is executed four more times in the execution of the

branching part of the algorithm. Finally, in the last array following S1, the final minimum is obtained.

Example 2. A2

GΔC
000XX11|x1
00X1X11|11
0X11X11|11
011XX11|1X
000X0X1|X1
00X10X1|11
0X110X1|11
011X0X1|11
000X00X|11
00X100X|11
0X1100X|11
011X00X|11
X110000|11

GΔD
GΔS
GΔZ

Here A2 is listed a GΔ array labelled A2 which is used in subsequent examples.

Example 3. Illustrates the structure of branching for the problem of Example 1: when the choice 21 is made, effectively, EBAR is executed twice, 21 designates cube 1x0|1x as the cube on which to branch. The column on the lower left corresponds to finding a solution which includes this cube in the solution, the column on the lower right is a computation of a solution which does not include it. (The final solution takes the one of the two of lower cost.)

Example 4. Consists of various examples of execution of several functions, such as S0 SHARP S2, etc. .

6. DEVELOPMENT OF THE PROGRAM

The programs for the multiple output two level minimization algorithm were written by the author during the summer of 1968. At the beginning of the summer, the author had no familiarity with the single output minimization algorithm, did not know APL, and had available a rough draft version of the algorithm in its F-notation formulation.

Among the factors contributing to the completion of the program were:

1) The use of an interactive computing terminal system, APL\360. The factor which the APL language itself contributed is hard to measure, but it is the author's feeling that it would not take appreciably longer in the assembly language, assuming that an interactive assembly language processor were at the same operational level as the APL interpreter. However, the fact that it is written in APL should make it more accessible to users.

2) The F-notation formulation, and the many clarifying conversations with Dr. J. P. Roth, developer of the algorithm. Once understood, this formulation provides a gestalt view of the algorithm.

3) The decision to model the program along the lines of the F-notation formulation, and to place first priority on completing the program and its documentation, at the possible cost of performance. This decision appeared to be justified, since the program was successfully run about a month before the end of the summer and a fairly extensive revision was made increasing

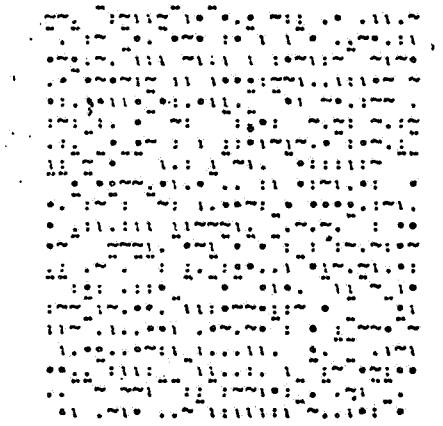
the performance by roughly a factor of 5.

The increases in efficiency were achieved by writing subroutines as array operations within a single statement.

Interfacing two cubes (XBX function) is a very straightforward application of this technique, where the array operations in APL are used as loop control. A more interesting illustration is the sharp product of two cubes (XIX function) where the initial program containing nested loop and subroutine calls has been combined within a single statement.

The use of single statements to loop increases the efficiency of program storage and execution time, but requires somewhat increased storage at run time. It is most worthwhile for the key operations such as sharping, and subsuming and the next candidate functions should be: XJX, XSX, SHARP , and XMX.

ROTH'S ALGORITHM IN APL\360



THE ATTACHED SET OF PROGRAMS IS REVISED AS OF 8/30/68

----------LEON S. LEVY

```
      ∇BBAR[□]∇
    ∇ C←BBAR A;B;D;E;F;G;H;I;J;K;L;M;N;O;P;Q;R;S
[1]   V←V CCAT A
[2]   'BBAR'
[3]   A
[4]   P←□
[5]   G←,(S←P XBV A)[1;]
[6]   →(1=∧/'X'=⍕,S[2;])/⍙2
[7]   H←,S[2;]
[8]   Q←(⍕G),'|',((⍕H)XBX ⍕H)
[9]   ⍙3:D←XMX((B←GΔC OF A)SHARP Q)CCAT H
[10]  I←XMX(E←GΔD OF A)CCAT(F←GΔS OF A)CCAT G
[11]  J←XMX F CCAT G
[12]  K←XMX((P←GΔZ OF A)MINUS Q)CCAT H
[13]  L←K CCAT GΔZ CCAT J CCAT GΔS CCAT I CCAT GΔD CCAT D CCAT GΔC
[14]  M←K CCAT GΔZ CCAT F CCAT GΔS CCAT E CCAT GΔD CCAT B CCAT GΔC
[15]  N←BBAR L
[16]  O←BBAR M
[17]  →((COST N)<COST O)/⍙1
[18]  C←O
[19]  →0
[20]  ⍙1:C←N
[21]  →0
[22]  ⍙2:H←⍳0
[23]  Q←G
[24]  →⍙3
      ∇
      ∇CCAT[□]∇
    ∇ C←A CCAT B
[1]   →(0=×/⍴A)/6
[2]   →(0=×/⍴B)/5
[3]   C←((⍴(XNX A))+(1 0)×⍴(XNX B))⍴((,B),,A)
[4]   →0
[5]   →(∧/,C=C←A)/0
[6]   →(∧/,C=C←B)/0
      ∇
      ∇COST[□]∇
    ∇ C←COST A;B;D;E;F;G
[1]   →(0=×/⍴A)/⍙63
[2]   A←XNX XAX A
[3]   F←1++/D×D←⍴B←A
[4]   F←0,(3⍴1),6⍴2
[5]   G←(D[1])⍴0
[6]   ⍙61:G[E]←F[1++/'1'=⍕+B[E;]]++/'X'=⍕+B[E;]
[7]   →(D[1]≥E←E+1)/⍙61
[8]   C←+/G
[9]   →0
[10]  ⍙63:C←0
      ∇
      ∇DELTA[□]∇
    ∇ C←E DELTA A;B;D;F;G;H;I;J;K;L;M;N
[1]   I←G←⍳0
[2]   B←0
[3]   E←XNX F
[4]   D←GΔZ OF A
[5]   ⍙51:H←XVX B[B;]SHARP XMX(GΔD OF A)CCAT D MINUS E[B←B+1;]
[6]   G←((⍕G)+E[B;]),'|',H)CCAT G
[7]   I←(E[B;]SHARP(XNX G)[B;])CCAT I
[8]   →(B<(⍴E)[1])/⍙51
[9]   J←XMX(GΔC OF A)SHARP G
[10]  L←XMX G CCAT GΔS OF A
[11]  K←XMX L CCAT GΔD OF A
[12]  M←XMX I CCAT(GΔZ OF A)MINUS E
[13]  C←M CCAT GΔZ CCAT L CCAT GΔS CCAT K CCAT GΔD CCAT J CCAT GΔC
      ∇
```

```
      ∇FBAR[□]∇
     ∇ C←FBAR A;□;D;E;F;G
[1]    V←V CCAT A
[2]    'FBAR'
[3]    A
[4]    →(0=×/⍴(B←GΔC OF A)SHARP E←GΔS OF A)/L1
[5]    →(1=(⍴GΔZ OF A)[1])/L3
[6]    A←(XTX A)XUX A
[7]    L3:→(0=×/⍴G←EXT A)/L2
[8]    C←FBAR G DELTA A
[9]    →0
[10]   L2:C←FBAR A
[11]   →0
[12]   L1:C←GΔS OF A
      ∇
      ∇EXT[□]∇
     ∇ C←EXT A
[1]    C←(GΔZ OF A)XAY XMX(GΔD OF A)CCAT GΔS OF A
      ∇
      ∇GΔCTEST[□]∇
     ∇ C←A GΔCTEST B;D
[1]    L72:C←⍳D←0
[2]    L71:C←XMX C CCAT A[1]TRIAL A[2]
[3]    →(((B>(⍴C)[1])∧10>D),(10<D←D+1))/(L71,L72)
      ∇
      ∇GΔDTEST[□]∇
     ∇ C←A GΔDTEST B;D;E;F;G;H;I
[1]    D←A GΔCTEST B
[2]    E←A GΔCTEST B
[3]    F←XMX E SHARP D
[4]    C←F INITIAL D
      ∇
      ∇IN[□]∇
     ∇ C←A IN B
[1]    C←(,∧/B=(⍴B)⍴A)\1
      ∇
      ∇INITIAL[□]∇
     ∇ C←A INITIAL B;D;E
[1]    GΔC←D⍴'GΔC',E←(D←(⍴XMX B)[2])⍴' '
[2]    GΔD←D⍴'GΔD',E
[3]    GΔS←D⍴'GΔS',E
[4]    GΔZ←D⍴'GΔZ',E
[5]    GΔ←D⍴'GΔ',E
[6]    GΔΔ←D⍴'GΔ',D⍴'*'
[7]    C←GΔZ CCAT GΔS CCAT A CCAT GΔD CCAT B CCAT GΔC
[8]    U←D-T←1+S←((,(XMX B)[1;])\'|')-1
[9]    L←S,C
[10]   V←1+\S++/Δ×Δ←0 1 0 1 0 0 0 0 0
[11]   P←'01X'
[12]   B←'0ΦΦ1101X'
[13]   E←'XΦ0ΦX1XXX'
[14]   H←'ΦΦΦΦΦΦ10Φ'
[15]   C←1 0 1 0 1 1 0 0 1
[16]   E←'01X'
      ∇
      ∇INTF[□]∇
     ∇ C←A INTF B
[1]    C←((S+A)XBX S+B),'|',(T+A)XBX T+B
      ∇
```

```
      ∇MINUS[⎕]∇
    ∇ C←A MINUS B;D;E;F
[1]   →((0=×/ρB)∨0=×/ρA)/X62
[2]   A←XNX A
[3]   M←(1+E←(ρA)[1])ρ0
[4]   D←1
[5]   X60:M[(XNX B)[D;]IN A]←1*
[6]   →((D←D+1)≤(ρXNX B)[1])/X60
[7]   C←A[(1-EοM)/ιE;]
[8]   →0
[9]   X62:C←A
    ∇
      ∇MXBAR[⎕]∇
    ∇ C←MXBAR A;B;D;E;F;G
[1]   V←ι0
[2]   →(0=×/ρ(B←GΔC OF A)SHARP E←GΔS OF A)/X1
[3]   D←SHARPALG XNX B CCAT GΔD OF A
[4]   C←XAX EBAR D CCAT A[ιGΔZ IN A;]
[5]   →0
[6]   X1:C←XAX E
    ∇
      ∇OF[⎕]∇
    ∇ C←A OF B;D;F
[1]   D←(∧/B=(ρB)ρA)ι1
[2]   B←((+/B=(ρB)ρGΔΔ)×(Dρ0),((ρB)[1]-D)ρ1)ι2
[3]   C←B[D+ιE-D+1;]
    ∇
      ∇PF[⎕]∇
    ∇ C←A PF B;D;E;F
[1]   C←D←0
[2]   →(0=(ρF←GΔS OF B)[1])×⍳1
[3]   ⍳1:→((D<(ρF)[1])∧1-C←C∨∧/A=S+F[D←D+1;])/⍳1
    ∇
      ∇RESID[⎕]∇
    ∇ M1←RESID M
[1]   M←XNX M
[2]   →(0=×/ρM)/X33
[3]   →((ρM)[1]=1)/X31
[4]   →((ρM)[1]=2)/X32
[5]   →(ρρM1)=ρρM1←((ρM)-(1 0))ρ((ρM)[2]φ,,M)
[6]   X31:→M1=M1←ι0
[7]   →0
[8]   X32:M1←((ρM)[2])ρ,M[2;]
[9]   →0
[10]  X33:M1←M
    ∇
      ∇SHARP[⎕]∇
    ∇ C←A SHARP B;D;E;F
[1]   →((0=ρA)∨0=ρB)/S3
[2]   →(XQX B)/S1
[3]   D←A SHARP B[1;]
[4]   C←D SHARP RESID B
[5]   →0
[6]   S1:→(XQX A)/S2
[7]   C←A XSX B
[8]   →0
[9]   S2:C←(,A)XJX,B
[10]  →0
[11]  S3:C←A
    ∇
```

```
      ∇SHARPALG[□]∇
    ∇ C←SHARPALG A;B;D;E
[1]   →(0=×/⍴A)/X34
[2]   A←XNX A
[3]   B←((⍴⍉↑,A[1;])⍴'X'),'|',((⍴⍉↑,A[1;])⍴'1')
[4]   D←B SHARP A
[5]   B←XMX D
[6]   D←B SHARP E
[7]   C←XMX D
[8]   →0
[9]   X34:C←A
      ∇
      ∇TRIAL[□]∇
    ∇ C←A TRIAL B;D;E;F;G;H;I
[1]   D←'0↓X'
[2]   E←'1↓X'
[3]   F←D[?A⍴3]
[4]   X70:G←E[?B⍴3]
[5]   →(1=∧/'X'=G)/X70
[6]   C←F,'|',G
      ∇
      ∇XAX[□]∇
    ∇ C←XAX A;B;D;E;F;G
[1]   →((XOX A)∨0=×/⍴A)/A1
[2]   →((⍴A)[1]=B←(⍉↑,A[1;])IN(XNX RESID A)[;1↓⍳B])/A2
[3]   C←XNX RESID A
[4]   C[B;]←(⍉↑C[B;]),(⍉↑C[B;])XBX ⍉↑A[1;]
[5]   C←XAX C
[6]   →0
[7]   A2:C←(XAX RESID A)CCAT A[1;]
[8]   →0
[9]   A1:C←A
      ∇
      ∇XAY[□]∇
    ∇ C←A XAY F;D;E
[1]   →(0=×/⍴A)/X35
[2]   A←XNX A
[3]   B←XNX B
[4]   D←1++/C←⍳0
[5]   Y3:→(0=×/⍴A[D;]SHARP(RESID(D-1)⌽[1]A)CCAT B)/Y1
[6]   Y2:→(((⍴A)[1])≥D←D+1)/Y3
[7]   →0
[8]   Y1:C←C CCAT A[D;]
[9]   →Y2
[10]  →0
[11]  X35:C←A
      ∇
      ∇XBX[□]∇
    ∇ C←A XBX B
[1]   C←⍉[(⍉↑B[⍳⍴A])÷3×⍳1+⍉⍳A[⍳⍴A]]
      ∇
      ∇XPY[□]∇
    ∇ C←B XPY A;D;E;F;G
[1]   C←(2,⍉D)⍴' '
[2]   F←,(GSS OF A)[B[1];]
[3]   F[B[2]]←'1',1∧/F≠F←⍉⍴'X'
[4]   C[1;]←(⍉↑E),F
[5]   →(0=×/⍴G←E XJX,C[1;])/Y1
[6]   C[2;]←G
[7]   →0
[8]   Y1:C[2;]←(⍉↑E),⍉⍴'X'
      ∇
```

```
      ∇XDX[□]∇
    ∇ C←A XDX B
[1]   C←⍕[(⍎⍕B[⍳⍴A])+3×¯1+⍎⍕A[⍳⍴A]]
    ∇
      ∇XEX[□]∇
    ∇ C←A XEX B
[1]   C←∧/⍕[(⍎⍕A[⍳⍴A])+3×¯1+⍎⍕B[⍳⍴A]]
    ∇
      ∇XIX[□]∇
    ∇ C←A XIX B;L;M;N
[1]   C←((⍕-N)⍴P)[((⍳⍴∘~∨/A[;M])×((⍳⍴'⍴')=P[;1])+N[M←(3×¯1+⍎⍕(P←N⍴(M⍴L⍴B)[;1]))+⍎⍕(M⍴L⍴A)[;1]]))/⍳⍕;]
    ∇
      ∇XJX[□]∇
    ∇ C←A XJX B;D;E;F;G
[1]   C←⍳0
[2]   D←∧/(E←(⍕+A)XDX ⍕+B)∈'X'
[3]   →('⍴'∈((⍕+A)XBX ⍕+B))/X5
[4]   →((⍕+B)XEX ⍕+A)/X6
[5]   →D/X16
[6]   C←(⍕+A),'|',F
[7]   X16:F←(⍕+A)XIX ⍕+B
[8]   G←0
[9]   X7:C←C,F[G←G+1;],'|',⍕+A
[10]  →(G<(⍴F)[1])/X7
[11]  C←(((⍴F)[1]+1-D),(⍴A))⍴C
[12]  →0
[13]  X5:C←A
[14]  →0
[15]  X6:→D/0
[16]  C←(⍕+A),'|',F
    ∇
      ∇XKX[□]∇
    ∇ C←A XKX B;D;E;F;G
[1]   C←((∧/G=F)∨(G←⍕+,A)XEX F←⍕+,B)∧(∧/E=D)∨(E←⍕+,B)XEX D←⍕+,A
    ∇
      ∇XMX[□]∇
    ∇ C←XMX A;B;D;E;F;G;H;I;J
[1]   A←XNX A
[2]   →(0=(⍴A)[1])/XM1
[3]   C←XNX A[J←J←1;]
[4]   X13:→((B←,(XNX C)[J;])XKX F←,A[J;])/X10
[5]   →(F XKX B)/X11
[6]   →X12
[7]   X10:→(J=J←J+I←1)/X14
[8]   X11:→(C[J;]=C[J;]←A[J;])/X10
[9]   X14:→((J≤D),J>D←(⍴A)[1])/(X13,0)
[10]  X12:→(I=(⍴C)[1])/X15
[11]  →(I=I←I+1)/X13
[12]  X15:C←C CCAT A[J;]
[13]  →(J=J←J+I←1)/X14
[14]  →0
[15]  XM1:C←A
    ∇
      ∇XNX[□]∇
    ∇ C←XNX A
[1]   →(2=⍴⍴A)/3
[2]   →(,C=C←(1,⍴A)⍴A)/0
[3]   C←A
    ∇
      ∇XOX[□]∇
    ∇ C←XOX A
[1]   C←(1=⍴⍴A)∨(⍴A)[1]=1
    ∇
```

```
      ∇XSX[□]∇
    ∇ C←A XSX B;D
[1]   →(0=×/⍴A)/⍙22
[2]   A←XNX A
[3]   B←,B
[4]   C←⍳D←0
[5]   ⍙21:C←C CCAT(,A[D←D+1;])XJX B
[6]   →(D<(⍴A)[1])/⍙21
[7]   C←XMX C
[8]   →0
[9]   ⍙22:C←A
    ∇
      ∇XTX[□]∇
    ∇ C←XTX A;B;D;E;F;G
[1]   C←(F←(⍴G←GΔZ OF A)[1],1)⍴0
[2]   →(0=×/⍴G)/⍙64
[3]   D←0
[4]   B←(3⍴1),6⍴2
[5]   ⍙40:→((⍳F←(GΔZ OF A)[D←D+1;])PP A)/⍙41
[6]   C[D;]←B[+/'1'=⍷+E]++/'X'=⍳+E
[7]   →(D<F)×⍙40
[8]   ⍙41:C[D;]←B[+/'1'=⍷+E]
[9]   →(D<F)×⍙40
[10]  →0
[11]  ⍙64:C←⍳0
    ∇
      ∇XUX[□]∇
    ∇ C←A XUX B;D;E;F;G;H;I;J;K;L;M;N;O
[1]   M←(E←(⍴K←GΔZ OF B)[1])⍴0
[2]   →((0=×/⍴K)∨1=E)/⍙48
[3]   J←GΔD OF B
[4]   D←1+N←1
[5]   ⍙44:→(∨/M[N,D])/⍙43
[6]   →('Φ'∈(⍳+G←K[N;])XBX ⍳+F←K[D;])/⍙43
[7]   →((I>N),(I=N),(I←A[N;])<N←A[D;])/(⍙47,⍙46,⍙45)
[8]   ⍙45:→(1-(G CCAT F)XVX J)/⍙43
[9]   →(M[D]←1)/⍙43
[10]  ⍙46:→(1-(F CCAT G)XVX J)/⍙45
[11]  →(M[N]←1)/⍙43
[12]  ⍙47:→(1-(F CCAT G)XVX J)/⍙43
[13]  →(M[N]←1)/⍙43
[14]  ⍙43:→(E>N←D=D←(N+1)⌈1+E|D+N=N←N+D=E)/⍙44
[15]  L←K[(1-N)/⍳E;]
[16]  O←GΔD CCAT(GΔC OF B)CCAT GΔC
[17]  C←L CCAT GΔZ CCAT(GΔS OF B)CCAT GΔS CCAT(GΔD OF B)CCAT O
[18]  →0
[19]  ⍙48:C←B
    ∇
      ∇XVX[□]∇
    ∇ C←A XVX B
[1]   C←0=×/,⍴(A[1;]SHARP B)SHARP A[2;]
    ∇
      ∇XWX[□]∇
    ∇ C←XWX A
[1]   →(XOX A)/⍙55
[2]   C←(⍳+A[1;])XBX XWX RESID A
[3]   →0
[4]   ⍙55:C←⍳+,A
    ∇
```

S0

GΔC
00X|1X
0X1|11
X11|11
11X|11
1X0|11
X00|1X
GΔD
GΔS
GΔZ

S1←MXBAR S0

EBAR

GΔC
00X|1X
0X1|11
X11|11
11X|11
1X0|11
X00|1X
GΔD
GΔS
GΔZ
1X0|11
X00|1X
11X|11
X11|11
00X|1X
0X1|11
EBAR

GΔC
X00|1X
1X0|1X
11X|1X
111|11
X11|1X
0X1|1X
00X|1X
GΔD
0X1|X1
1X0|X1
GΔS
1X0|X1
0X1|X1
GΔZ
1X0|1X
0X1|1X
00X|1X
X11|11
11X|11
X00|1X
EBAR

GΔC
X00|1X
1X0|1X
11X|1X
111|11
X11|1X
0X1|1X
00X|1X
GΔD
0X1|X1
1X0|X1
GΔS
1X0|X1
0X1|X1
GΔZ
1X0|1X
0X1|1X
00X|1X
X11|11
11X|11
X00|1X
EBAR

2

GΔC
GΔD
11X|11
X00|1X
0X1|1X
1X0|X1
0X1|X1
GΔS
X00|1X
11X|11
0X1|1X
1X0|X1
0X1|X1
GΔZ
1X0|1X
EBAR

GΔC
X00|1X
1X0|1X
11X|1X
111|11
X11|1X
0X1|1X
00X|1X
GΔD
0X1|X1
1X0|X1
GΔS
1X0|X1
0X1|X1
GΔZ
X00|1X
11X|11
X11|11
00X|1X
1X0|1X
EBAR

GΔC
111|X1
1X0|1X
GΔD
00X|1X
X11|1X
1X0|X1
0X1|X1
GΔS
X11|1X
00X|1X
0X1|X1
1X0|X1
GΔZ
X11|X1
1X0|1X
11X|11
X00|1X
EBAR

GΔC
111|X1
GΔD
1X0|1X
0X1|X1
1X0|X1
X11|1X
00X|1X
GΔS
1X0|1X
1X0|X1
0X1|X1
00X|1X
X11|1X
GΔZ
11X|11
X11|X1
EBAR

GΔC
GΔD
X11|X1
00X|1X
X11|1X
1X0|X1
0X1|X1
1X0|1X
GΔS
X11|X1
X11|1X
00X|1X
0X1|X1
1X0|X1
1X0|1X
GΔZ

S1

X11|11
00X|1X
0X1|X1
1X0|11

A2

GΔC
000XX11|X1
00X1X11|11
0X11X11|11
011XX11|1X
000X0X1|X1
00X10X1|11
0X110X1|11
011X0X1|11
000X00X|11
00X100X|11
0X1100X|11
011X00X|11
X110000|11
GΔD
GΔS
GΔZ

```
GΔC
00X|1X
0X1|1X
111|11
X11|1X
GΔD
1X0|X1
0X1|X1
1X0|1X
GΔS
1X0|X1
0X1|X1
1X0|1X
GΔZ
00X|1X
X11|11
11X|11
X00|1X
0X1|1X
EBAR

GΔC
GΔD
X11|11
00X|1X
1X0|1X
0X1|X1
1X0|X1
GΔS
00X|1X
X11|11
1X0|1X
0X1|X1
1X0|X1
GΔZ
0X1|1X
EBAR

GΔC
00X|1X
0X1|11
X11|11
11X|11
1X0|11
X00|1X
GΔD
GΔS
GΔZ
00X|1X
0X1|11
X11|11
11X|11
1X0|11
X00|1X
```

```
EBAR

GΔC
X00|1X
1X0|1X
11X|1X
X11|1X
111|11
0X1|1X
00X|1X
GΔD
1X0|X1
0X1|X1
GΔS
0X1|X1
1X0|X1
GΔZ
0X1|1X
1X0|1X
X00|1X
11X|11
X11|11
00X|1X
BBAR

GΔC
X00|1X
1X0|1X
11X|1X
X11|1X
111|11
0X1|1X
00X|1X
GΔD
1X0|X1
0X1|X1
GΔS
0X1|X1
1X0|X1
GΔZ
0X1|1X
1X0|1X
X00|1X
11X|11
X11|11
00X|1X
□:
```



```
EBAR

GΔC
X00|1X
1X0|1X
11X|1X
X11|1X
111|11
0X1|1X
00X|1X
GΔD
1X0|X1
0X1|X1
GΔS
0X1|X1
1X0|X1
GΔZ
00X|1X
X11|11
11X|11
X00|1X
0X1|1X
EBAR

GΔC
0X1|1X
111|X1
GΔD
X00|1X
11X|1X
0X1|X1
1X0|X1
GΔS
11X|1X
X00|1X
1X0|X1
0X1|X1
GΔZ
11X|X1
0X1|1X
X11|11
00X|1X
```

```
EBAR

GΔC
111|X1
GΔD
0X1|1X
1X0|X1
0X1|X1
11X|1X
X00|1X
GΔS
0X1|1X
0X1|X1
1X0|X1
X00|1X
11X|1X
GΔZ
X11|11
11X|X1
EBAR

GΔC
GΔD
11X|X1
X00|1X
11X|1X
0X1|X1
1X0|X1
0X1|1X
GΔS
11X|X1
11X|1X
X00|1X
1X0|X1
0X1|X1
0X1|1X
GΔZ
```

```
)GRP EXAMPLES
```

C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	S1	S0	S2	S3	S4
S5	S6	S7	S8	S9	V0	V1	V3	V4	V5	D	U	T	E	L
U	A	E	B	E	H	G	GΔC	GΔD	GΔS	GΔZ	GΔ	GΔΔ	M	NULL

```
      C0
100X1
      C1
XXXXX
      C2
1100X
      C3
1010X
      C4
1XX01
      C5
01101
      C6
01XX1
      C7
X100X
      C8
XXX10
      C9
11111
      S0
X0110|X1
      S1
0X110|1X
      S2
XX11X|11
      S3
1X010|1X
      S4
1XX10|11
      S5
XXX00|X1
      S6
1XX10|X1
      S7
01X01|X1
      S8
00101|1X
      S9
110X1|11
      V0

0XX1X|X1
1X0XX|X1
0111X|1X
```

```
      V4 MINUS S3

01X01|X1
110X1|11
      V4 MINUS S2

01X01|X1
00101|1X
110X1|11
      S2 MINUS NULL
XX11X|11
      S2 MINUS S3

XX11X|11
      S2 MINUS S2

      S2 MINUS V2

      S2 MINUS V3

XX11X|11
      RESID V3

1XX10|11
XXX00|X1
1XX10|X1
      RESID RESID V3

XXX00|X1
1XX10|X1
      RESID RESID RESID V3
1XX10|X1
      RESID RESID RESID ES
                        ^
                        RESID V3

      RESID RESID RESID RESID RESID V3

      RESID S0

      RESID NULL

      S0 SHARP S1

 X0110|X1
 10110|X1
      S0 SHARP S2

      S2 SHARP S0

 XX11X|1X
 X111X|11
 XX111|11
      S0 SHARP NULL
 X0110|X1
      NULL SHARP S0
 VALUE ERROR
      NULL SHARP S0
                 ^
      NULL SHARP S0
```

```
      B
3  2
      C←3 TRIAL 2
      C
0X1|11
      NULL INITIAL C

GΔC
0X1|11
GΔD
GΔS
GΔZ
      V0 INITIAL V1

GΔC
1X00X|11
01110|11
1X011|1X
0010X|X1
GΔD
0XX1X|X1
1X0XX|X1
0111X|1X
GΔS
GΔZ
      S0 IN V2
1
      S1 IN V2
2
      S2 IN V2
3
      S3 IN V2
4
      S4 IN V2
4
      S0 IN V4
4
      S0 IN V3
5
      S0 INTP S0
X0110|X1
      S0 INTP S1
00110|11
      S0 INTP S2
X0110|11
      S0 INTP S3
10010|11
      S0 INTP S4
10110|11
      V2 MINUS NULL

X0110|X1
0X110|1X
XX11X|11
      NULL MINUS V2

      V5 MINUS V3

X0110|X1
0X110|1X
XX11X|11
01X01|X1
00101|1X
110X1|11
```

```
      GΔΔ
GΔΣ
      GΛ
GΔ
      GΔΔ
GΔ******
      N

 100X1
 XXXXX
 1100X
 1010X
 1XX01
 01101
 01XX1
 X100X
 XXX10
 11111
      NULL

      S0 CCAT V0

 0XX1X|X1
 1X0XX|X1
 0111X|1X
 X0110|X1
      V0 CCAT V1

 1X00X|11
 01110|11
 1X011|1X
 0010X|X1
 0XX1X|X1
 1X0XX|X1
 0111X|1X
      S0 CCAT NULL
 X0110|X1
      C0 CCAT C1 CCAT C2 CCAT C3

 1010X
 1100X
 XXXXX
 100X1
      A←5 2
      A
5  2
      A GΔCTEST 3

 1101X|1X
 00X10|X1
 10X10|11
      A GΔCTEST 1

 11X00|11
      A GΔCTEST 2

 GΔC
 011X0|11
 00X10|X1
 GΔD
 0X111|11
 0X000|11
 GΔC
 GΔZ
      B←3 2
```

```
      V1

1X00X|11
01110|11
1X011|1X
0010X|X1
      V2

X0110|X1
0X110|1X
XX11X|11
      V3

1XC10|1X
1XX10|11
XXX00|X1
1XX10|X1
      V4

01X01|X1
00101|1X
110X1|11
      V5

X0110|X1
0X110|1X
XX11X|11
1XC10|1X
1XX10|11
XXX00|X1
1XX10|X1
01X01|X1
00101|1X
110X1|11
      D
0
      U
2
      T
6
      S
5
      L
5  5
      N
0  1  2  3  4
      A
0  1  0  1  0  0  0  0
      L
01X
      E
0¢0¢011C1X
      E
X¢0¢X1XXX
      E
¢¢¢¢¢¢10¢
      E
1  0  1  0  1  1  0  0
      GΔC
GΔC
      GΔD
GΔ
      GΔS
GΔS
```

EBAR

GAC
000XX11|X1
00X1X11|11
0X11X11|11
011XX11|1X
000X0X1|X1
00X10X1|11
0X110X1|11
011X0X1|11
000X00X|11
00X100X|11
0X1100X|11
011X00X|11
X110000|11
GAD
GAS
GAZ
011X00X|11
011X0X1|11
0X1100X|11
0X110X1|11
011XX11|1X
0X11X11|11
000X00X|11
000X0X1|X1
00X100X|11
00X10X1|11
00X1X11|11
000XX11|X1
X1100C0|11
EBAR

GAC
0X1100X|1X
001100X|11
0X11000|11
011X001|1X
GAD
X110000|11
000XX11|X1
00X1X11|1X
000X00X|11
0X11X11|X1
011XX11|1X
011X0X1|X1
GAS
011X0X1|X1
011XX11|1X
0X11X11|X1
000X00X|11
00X1X11|1X
000XX11|X1
X110000|11
GAZ
011X0X1|1X
0X11X11|1X
00X1X11|X1
00X10X1|11
00X100X|11
000X0X1|X1
0X110X1|11
0X1100X|11

EBAR

GAC
0110001|1X
GAD
0X1100X|11
011X0X1|X1
011XX11|1X
0X11X11|X1
000X00X|11
00X1X11|1X
000XX11|X1
X110000|11
GAS
0X1100X|11
X110000|11
000XX11|X1
00X1X11|1X
000X00X|11
0X11X11|X1
011XX11|1X
011X0X1|X1
GAZ
011X00X|11
00X1X11|X1
011X0X1|1X
EBAR

GAC
GAD
011X0X1|1X
X110000|11
000XX11|X1
00X1X11|1X
000X00X|11
0X11X11|X1
011XX11|1X
011X0X1|X1
0X1100X|11
GAS
011X0X1|1X
011X0X1|X1
011XX11|1X
0X11X11|X1
000X00X|11
00X1X11|1X
000XX11|X1
X110000|11
0X1100X|11
GAZ
00X1X11|X1

011X0X1|11
011XX11|1X
0X11X11|X1
000X00X|11
00X1X11|1X
000XX11|X1
X110000|11
0X1100X|11

```
      G1←A GADTEST 3
      G1

GAC
XXXXX|X1
111XX|1X
X110X|1X
GAD
X0XXX|1X
1X010|1X
GAS
GAZ
      G2←A GADTEST 4
      G2

GAC
X10X0|11
1XX0X|11
0007G|X1
0001X|11
GAD
0X1XX|X1
XX11X|X1
10010|X1
0X1X0|11
XX110|11
GAS
GAZ
      G3←NULL INITIAL V0
      G3

GAC
0XX1X|X1
1X0XX|X1
0111X|1X
GAD
GAS
GAZ
      G4←NULL INITIAL V1
      G4

GAC
1X00X|11
01110|11
1X011|1X
0010X|X1
GAD
GAS
GAZ
      G5←NULL INITIAL V3
      G5

GAC
1X010|1X
1XX10|11
XXX00|X1
1XX10|X1
GAD
GAS
GAZ
```

```
      ρS0
8
      ρXXX S0
1  8
      ρV5
10  8
      ρV3
4  8
      ρXNX V3
4  8
      XOX NULL
1
      XOX S0
1
      XOX XNX S0
1
      XOX V3
0
      XOX V3
1
      NULL XSX S6

      NULL XSX V3

      S3 XSX S2

1X010|1X
      S3 XSX S3

      V3 XSX S2

XXX00|X1
1X010|11
      V3 XSX S9

XXX00|X1
1XX10|11
      V6←S1 CCAT S0
      V6

X0110|X1
0X110|1X
      V6 XVX NULL
0
      V6 XVX S1
0
      V6 XVX V6
1
      V6 XVX S3
0
      V6 XVX V3
0
      XNX S0
X1
      XNX S9
11
      XNX V3
11
      XNX V0
11
```

```
XXXX0

      C1 XIX C0

0XXXX
X1XXX
XX1XX
XXXX0
      C1 XIX C2

0XXXX
X0XXX
XX1XX
XXX1X
      C2 XIX C3

      C3 XIX C2

      S0 XJX S1

X0110|X1
10110|X1
      S1 XJX S0

0X110|1X
01110|1X
      S3 XJX S4

      S4 XJX S3

1XX10|X1
1X110|11
      S0 XKX S0
1
      S0 XKX S1
0
      S1 XKX S0
0
      S1 XKX S2
0
      S2 XKX S1
1
      XMX NULL

      XMX S3

1X010|1X
      XMX V5

110X1|11
00101|1X
01X01|X1
XXX00|X1
1XX10|11
XX11X|11
X0110|X1
      ρNULL
0
      XNX NULL

      ρXNX NULL
0
      XNX S0

X0110|X1
```

```
      NULL XAY NULL

      NULL XAY S3

      S2 XAY NULL
XX11X|11
      S2 XAY S2

      S2 XAY S3
XX11X|11
      S3 XAY S2
1X010|1X
      V3 XAY NULL

XXX00|X1
1XX10|11
      V3 XAY S0

XXX00|X1
1XX10|11
      V3 XAY S3

XXX00|X1
1XX10|11
      V3 XAY V4

XXX00|X1
1XX10|11
      V4 XAY V3

110X1|11
00101|1X
01X01|X1
      C0 XBX C1
100X1
      C1 XBX C0
100X1
      C2 XBX C3
1ΦΦ0X
      C3 XPX C2
1ΦΦ0Y
      C0 XDX C1
100Y1
      C1 XDX C0
XXXXX
      C8 XDX C9
XXXXΦ
      C9 XDX C9
XXXXX
      C0 XFX C1
0
      C1 XFX C1
1
      C5 XFX C6
0
      C6 XFX C5
1
      C0 XIX C1

      C1 XIX C0

0XXXX
X1XXX
XX1XX
```

```
      V2 SHARP S3

XX11X|11
      S3 SHARP V3

      V3 SHARP S3

1X110|11
XXX0C|X1
1XX10|X1
      S3 SHARP S3 MINUS S2

      S3 SHARP S3 MINUS S3
1X010|1X
      SHARPALG NULL

      SHARPALG S0

X0110|X1
      SHARPALG V3

1XXX0|X1
XXX00|X1
1XX10|11
      SHARPALG V5

011XX|X1
XX11X|11
XX1X0|X1
001X1|1X
110XX|X1
11X1X|11
1XXX0|X1
110X1|11
1XX10|11
C1X0X|X1
X100X|X1
XXX00|X1

      XAX NULL

      XAX S0
X0110|X1
      XAX V3

1X010|1X
XXX00|X1
1XX10|11
      XAX V5

X0110|X1
0X110|1X
XX11X|11
1X010|1X
XXX0C|X1
1XX10|11
01X01|X1
00101|1X
110X1|11
```

MXBAR G4

EBAR

GAC
1X00X|11
01110|11
1X011|1X
0010X|X1
GAD
GAS
GAZ
1X00X|11
0010X|X1
1X0X1|1X
01110|11
EPAP

GAC
GAD
01110|11
1X0X1|1X
0010X|X1
1X00X|11
GAS
1X00X|11
0010X|X1
1X0X1|1X
01110|11
GAZ

1X00X|11
0010X|X1
1X0X1|1X
01110|11
)SAVE CONTINUE
29.43.01 08/28/68

MXBAR G3

EBAR

GAC
0XX1X|X1
1X0XX|X1
0111X|1X
GAD
GAS
GAZ
0XX1X|X1
XX01X|X1
0111X|11
1X0XX|X1
EBAR

GAC
GAD
1X0XX|X1
0111X|1X
0XX1X|X1
GAS
0XX1X|X1
0111X|1X
1X0XX|X1
GAZ
0111X|X1
XX01X|X1

0XX1X|X1
0111X|1X
1X0XX|X1
)SAVE CONTINUE
29.26.46 08/28/68
I21
6576

MXBAR G2

EPAR

GAC
X10X0|11
1XX0X|11
000X0|X1
0001X|11
GAD
0X1XX|X1
XX11X|X1
10010|X1
0X1X0|11
XX110|11
GAS
00X1X|X1
XX1XX|X1
0XX10|11
0001X|11
XXXX0|X1
X1XX0|11
XX1X0|11
1XX0X|11
EPAR

GAC
GAD
1XX0X|11
X1XX0|1X
0001X|11
XX110|11
0X1X0|11
XXXX0|X1
XX11X|X1
0X1XX|X1
GAS
0001X|11
XXXX0|X1
X1XX0|1X
1XX0X|11
GAZ
X1XX0|X1
0XX10|11
XX1XX|X1
00X1X|X1

0001X|11
XXXX0|X1
X1XX0|1X
1XX0X|11

MXBAR G1

EPAR

GAC
XXXXX|X1
111XX|1X
X110X|1X
GAD
X0XXX|1X
1X010|1X
GAS
GAZ
XXXXX|X1
X0XXX|11
1X1XX|11
1XX10|11
XX10X|11
EPAR

GAC
GAD
XX10X|1X
1X1XX|1X
XXXXX|X1
1X010|1X
X0XXX|1X
GAS
XXXXX|X1
1X1XX|1X
XX10X|1X
GAZ
1X1XX|X1
XX10X|X1
1XX10|11

XXXXX|X1
1X1XX|1X
XX10X|1X

MXBAR G0

EPAR

GAC
XX11X|1X
X0111|11
X0X11|1X
0XX1X|11
00100|X1
GAD
1X000|1X
01101|11
11101|1X
0000X|11
GAS
GAZ
000XX|11
0XX1X|11
00XX0|X1
011X1|11
XX11X|1X
X0X11|1X
X0111|11
X0000|1X
X11X1|1X
1X000|1X
EPAR

GAC
GAD
X0111|X1
X0X11|1X
XX11X|1X
00XX0|X1
0XX1X|11
0000X|11
11101|1X
01101|11
1X000|1X
GAS
0XX1X|11
00XX0|X1
XX11X|1X
X0X11|1X
X0111|X1
GAZ
X0111|1X
1X000|1X

0XX1X|11
00XX0|X1
XX11X|1X
X0X11|1X
X0111|X1

GAMMA GADT
G0

GAC
XX11X|1X
X0111|11
X0X11|1X
0XX1X|11
00100|X1
GAD
1X000|1X
01101|11
11101|1X
0000X|11
GAS
GAZ
GAC OF G0

XX11X|1X
X0111|11
X0X11|1X
0XX1X|11
00100|X1
GAD OF G0

1X000|1X
01101|11
11101|1X
0000X|11
GAS OF G0

GAZ OF G0

GAS IN G0
12

```
      EXT I

1X0|11
0X1|11

      XTX H

  1
  1
  3
  3
  3
  3

      (XTX H) XUX H

GΔC
X00|1X
1X0|1X
11X|1X
X11|1X
111|11
0X1|1X
00X|1X
GΔD
1X0|X1
0X1|X1
GΔS
0X1|X1
1X0|X1
GΔZ
0X1|1X
1X0|1X
X00|1X
11X|11
X11|11
00X|1X
```

```
                H

GΔC
X00|1X
1X0|1X
11X|1X
X11|1X
111|11
0X1|1X
00X|1X
GΔD
1X0|X1
0X1|X1
GΔS
0X1|X1
1X0|X1
GΔZ
0X1|1X
1X0|1X
X00|1X
11X|11
X11|11
00X|1X
                I

GΔC
00X|1X
0X1|11
X11|11
11X|11
1X0|11
X00|1X
GΔD
GΔS
GΔZ
00X|1X
0X1|11
X11|11
11X|11
1X0|11
X00|1X
```

```
      MXBAR MILLER
EBAR

GΔC
XX010|1
X1111|1
XX001|1
00X11|1
X0X00|1
10X11|1
GΔD
X0011|1
X1001|1
11010|1
10100|1
GΔS
GΔZ
X00XX|1
X0X00|1
X0X11|1
XX111|1
XX010|1
XX001|1
EBAR

GΔC
X1001|1
GΔD
XX111|1
X0X00|1
XX010|1
X1001|1
X00XX|1
GΔS
X00XX|1
X0X00|1
XX111|1
XX010|1
GΔZ
BBAR

GΔC
X1001|1
GΔD
XX111|1
X0X00|1
XX010|1
X1001|1
X00XX|1
GΔS
X00XX|1
X0X00|1
XX111|1
XX010|1
GΔZ
□:
      1 1
INDEX ERROR
XBY[2] E←,(GΔZ OF A)[B[1];]
          ^
      A
```

```
      MILLER

GΔC
XX010|1
X1111|1
XX001|1
00X11|1
X0X00|1
10X11|1
GΔD
X0011|1
X1001|1
11010|1
10100|1
GΔS
GΔZ
```

```
                MXBAR G5
EBAR

GΔC
1X010|1X
1XX10|11
XXX00|X1
1XX10|X1
GΔD
GΔS
GΔZ
XXX00|X1
1XXX0|X1
1XX10|11
EBAR

GΔC
GΔD
1XX10|11
XXX00|X1
GΔS
XXX00|X1
1XX10|11
GΔZ
1XXX0|X1

XXX00|X1
1XX10|11
```